

Personal Computer

PC
NEC

アセンブリ言語

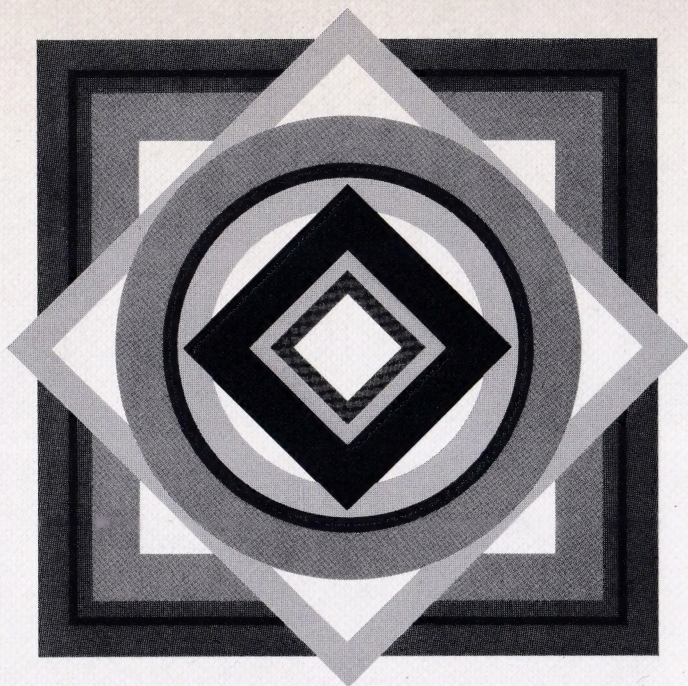
プログラミング入門[1]

川村清一 著

ナツメ社

9801





Personal Computer

PC
NEC

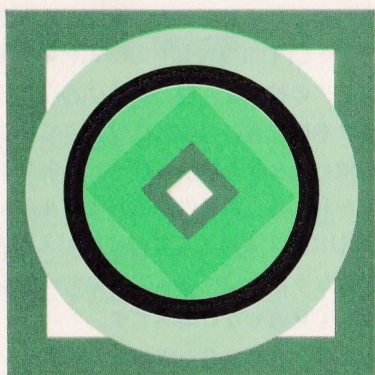
9801

アセンブリ言語

プログラミング入門[1]

川村清一 著

ナツメ社



はじめに

16ビットのCPUを搭載した機種が各社から発表されるにつれて、パーソナル・コンピュータも16ビット全盛時代をむかえつつあります。

なかでも特に注目を浴びたのが、我国で多くのベストセラー商品を出し続けて来た日本電気による、PC-9801の発表でした。PC-9801は私達の予想、すなわち当然のことながら8088をCPUとして採用するであろう予想を見事に裏切り、はじめから本格的な8086をCPUとして搭載していたのです。

8086は日本電気のパーソナル・コンピュータPC-9801に採用されたことにより、その機能と操作性を高く評価され、今や我国の16ビットCPU市場においてシェアの大半を占めるようになりました。

シェアの拡大は、別の面では8086のアセンブリ言語が、一部のマニアや研究者のためだけのものではなく、広く一般のユーザにも手軽に使いこなせる段階に移行して来たことを意味します。

しかし、8086の浸透があまりにも急速であったためもあり、このCPUに関するほとんどの文献が入門書というよりは、8086の評価を行なうためのマニュアルやデータ・シートに類するものばかりであることも見逃せない事実です。

本書は、8086を媒体としてアセンブリ言語の基礎を学びたい皆さんのために、短いサンプル・プログラムを数多くあげて、平易な解説を行なうものです。使用機種としては最も一般的であろうPC-9801を想定していますが、全てのプログラムに対して実行結果が示してありますので、機械をお持ちでない方でも自然に読み進むことができますと思います。

この本が、8086のアセンブリ言語を理解・評価されるための一助となれば幸いこれに過ぎるものはありません。

なお、本書の執筆および出版に当たり一方ならぬお世話になりました、田村正隆氏、小牧自行氏、沼田裕氏、脇英世助教授（東京電機大学）に感謝いたします。

FORESIGHT企画部代表 川村 清

もくじ

第0章 アセンブリ言語の入門にあたって ————— 7

- 0・0 アセンブリ言語とは？ ——— 8
- 0・1 準備するもの ——— 10
- 0・2 フローチャート用紙とコーディング用紙 ——— 11
- 0・3 数の豆知識 ——— 12

第1章 PC-9801の基礎知識 ————— 17

- 1・0 PC-9801のOS ——— 18
- 1・1 N₈₈-BASICのキャラクタ ——— 19
- 1・2 PC-9801のメモリ・マップ ——— 20
- 1・3 メモリのセグメンテーション ——— 22
- 1・4 機械語モニタの概要 ——— 24
- 1・5 機械語モニタの起動方法 ——— 26

第2章 8086の基礎知識 ————— 29

- 2・0 8086の生い立ち ——— 30
- 2・1 8086の設計思想 ——— 33
- 2・2 8086のレジスタ・セット ——— 34
- 2・3 セグメント・レジスタの役割 ——— 36
- 2・4 汎用レジスタの役割 ——— 38
- 2・5 命令ポインタの役割 ——— 39
- 2・6 ポインタとインデックス・レジスタの役割 ——— 40
- 2・7 フラグの役割 ——— 41

第3章 簡単なプログラム ————— 43

- 3・0 メモリの内容をコピーするプログラム ——— 44
- 3・1 PC-9801の画面構成 ——— 52
- 3・2 メッセージを表示するプログラム ——— 58

3・3	8ビットの加算を行なうプログラム	62
3・4	16ビットの加減算を行なうプログラム	65
3・5	算術演算と論理演算	72
3・6	論理演算をテストするプログラム	76
3・7	16ビットデータを10倍するプログラム	82
3・8	サブルーチンの使い方	85
3・9	スタックの概念とレジスタ退避	90
3・10	レジスタ退避の使用方法	97
3・11	レジスタ退避の応用例	98
3・12	サブルーチン・コールとレジスタ退避	101
3・13	8086の乗除算命令	102
3・14	乗除算命令を使ったプログラム	104
3・15	第3章のまとめ	108

第4章 分岐のあるプログラム 109

4・0	分岐のあるプログラム	110
4・1	ゼロ・フラグ (ZF) による分岐	113
4・2	キャリ・フラグ (CF) による分岐	118
4・3	分岐の組み合わせ	122

第5章 くり返しのあるプログラム 127

5・0	くり返し処理のあるプログラム	128
5・1	基本的なループ処理	130
5・2	画面操作への応用—I	135
5・3	画面操作への応用—II	137
5・4	画面操作への応用—III	140
5・5	画面操作への応用—IV	144
5・6	簡単なデモンストレーション	147
5・7	ローテートおよびシフトの概念	150
5・8	ローテート・シフト命令を使った乗算方法	152

-
- 5・9 割り込み（インタラプト）の基礎知識——156
 - 5・10 本文を読み終えた皆さんに——158

第6章 8086のインストラクション・セット——159

- 6・0 8086のインストラクション・セット——160
- 6・1 データ転送命令——162
- 6・2 算術演算命令——165
- 6・3 乗除算命令——168
- 6・4 論理演算命令——170
- 6・5 ストリング操作命令——172
- 6・6 分岐命令——175
- 6・7 CPU制御命令——179

第7章 付録: μ PD8086インストラクション活用表——181

- 7・0 8086のインストラクション活用表——182
- 7・1 データ転送命令——186
- 7・2 算術演算命令——190
- 7・3 乗除算命令——194
- 7・4 論理演算命令——196
- 7・5 ストリング操作命令——206
- 7・6 分岐命令——208
- 7・7 CPU制御命令——214

0章

アセンブリ言語の 入門にあたって

0

0・0 アセンブリ言語とは？

アセンブリ言語

高級言語

アセンブリ言語とは難しいものなのでしょうか？

BASICなどの高級言語を相当高度に使いこなしている人でも、アセンブリ言語は難しいと考えて敬遠している場合があります。実際にアセンブリ言語や機械語に対する批評としてよく耳にするのも「難しい」とか「わからない」などという否定的な意見が大半です。

それでは、アセンブリ言語は本当に難しいものなのでしょうか？

本当にわからないものなのでしょうか？

否、私はアセンブリ言語が難しいものだとは思いません。しかし、アセンブリ言語が「多少手間のかかる」ものであることは確かです。

コンピュータのプログラムをつくる場合、どんな手段を使うにしろ、多かれ少なかれ手間がかかることは否めません。一時期流行した簡易言語のたぐいを利用した場合にさえ多少の手間はついて回ります。

私には、この「手間」ということと「難しさ」ということを混同して、アセンブリ言語を難しいと思い込んでいる人が多いように思えるのですが、皆さんはどう思われるのでしょうか？

もし、皆さんが「アセンブリ言語は難しい」と思い込んでおられるなら、今すぐにでも「アセンブリ言語は手間がかかる」と考えなおしていただきたいと思います。

ところで、この手間のかかるアセンブリ言語とは一体何なのでしょうか？

皆さんも良くご承知のように、コンピュータは0と1の電気信号しか理解できません。したがって、コンピュータが直接理解できる形でプログラムを書くためには、0と1だけしか使ってはいけないことになります。この、0と1のみによって構成されたプログラムを機械語（マシン語）プログラムといい、普通は数字の0～9とアルファベットのA～Fを用いた16進数で記述します。

3C 03 53 51 52 C5 1F 8B 4F 02

ところが、一見して数値のら列でしかない機械語プログラムは、人間にとって、たいへんわかりにくいので、もう少しわかりやすいように、各命令の動作に相当する英単語を適当に省略(変形)した形のプログラムが要求されます。

これが、本書のタイトルにもなっているアセンブリ言語のプログラムです。

機械語（マシン語）

CMP	AL,03H
PUSH	BX
PUSH	CX
PUSH	DX
LDS	BX,[BX]
MOV	CX,2[BX]

ちなみに、上図のCMPはCOMPARE（比較する）、PUSHはそのま
まPUSH（押す）の意味です。このように英単語と対応させれば覚え
たりプログラムを読んだりするのも楽ですし、学生の方は英単語の勉
強もできて1石2鳥ではないでしょうか？

このアセンブリ言語のプログラムから、機械語のプログラムに変換
する作業が、特に命令数の多い8086（ μ PD8086）の場合に
は大変な作業になるのですが、この変換を自動的に行ってくれるプ
ログラムを、アセンブラと言います。逆に機械語のプログラムからア
センブリ言語のプログラムに変換するものを、ディス・アセンブラ（逆
アセンブラ）と言います。

アセンブラ
ディス・アセンブラ

ところで、なぜアセンブリ言語でのプログラミングを行なう必要が
あるのでしょうか？

本書が、アセンブリ言語の入門書であることを知った上で購入され
た皆さんには、この質問に対する解答が十分にわかっていることと思
います。

BASICのような高級言語によるプログラムは一般に実行速度が遅く、
メモリの利用効率もあまり良くありません。また、入出力命令の拡張
性が乏しいことも事実です。これに比較してアセンブリ言語のプログ
ラムは非常に高速で、ゲームや制御のように瞬間的な応答を必要とす
る場合や、微妙なコントロールを必要とする場合にはなくてはならな
いものですし、メモリやハードウェアを意識した、きめの細かいソフ
トウェアをつくることも可能です。

より手間をかけた方が、すみずみまで気の行きとどいたきめの細か
いものが完成するという点で、コンピュータのプログラミングは、た
えば書籍の出版などと非常に似ているような気がします。

もちろん、すべてのプログラムをアセンブリ言語で書く必要はあり
ませんが、BASICなどと組み合わせることにより非常に豊かな可能性
が生まれて来ると思います。

0・1 準備するもの

本書では、皆さんが次のシステムを使用しているものと仮定して話を進めて行きます。

PC-9801本体

まず、PC-9801本体、これは言うまでもなく必要ですが、本体の他に、CRTディスプレイとディスク・ユニットが絶対に必要です。

CRTディスプレイ

CRTディスプレイは、PC-9801専用の高解像度ディスプレイ(640×400ドットのフル・グラフィックが楽しめる)やカラー・ディスプレイでない、普通のグリーン・ディスプレイなどでかまいませんが、できれば80桁×25行(2000文字)の表示が可能なのが使いやすいと思います。

ディスク・ユニット

ディスク・ユニットに関しては、なぜ必要なのか疑問に思われるかも知れませんが、PC-9801の内蔵モニタではディスク・ベースでないアセンブルおよび逆アセンブルを行なうためのコマンド(命令)が使用できないため、ディスクなしでは話になりません。最近ではディスク・ユニット(特にミニ・ディスク・ユニット)の価格が低下してきたことでもありますし、PC-9801ではCMTインタフェースがオプション(別売)であることも考えますと、ディスク・ユニットをお持ちでない皆さんも、1台購入しておいて絶対に損ではないと思います。もちろん、使用するディスク・ユニットに適合したシステム・ディスクも用意しておいてください。

システム・ディスク

以上のシステムが準備されていれば一応アセンブリ言語の学習が進められると思いますが、プリンタがあると非常に便利です。グラフィックが使えるものや漢字が打ち出せるものなどもありますが、学習用ということであれば、英数字および記号(カッコなど)ぐらいが使用できれば十分です。CRTディスプレイ上で表示させるだけでも相当のことができますが、こうしたソフト・コピーというやり方では1画面分しか見ることができないので、1画面におさまらないような少し長いプログラムになると、デバッグなどもたいへんです。

最後に、本書では使用しませんが、アセンブリ言語の学習をある程度進めて行くと内蔵のアセンブラ/逆アセンブラでは不満が出てくるでしょう。その時には市販されているアセンブラを購入することになると思いますが、ぜひ御自分に合った評判の良いアセンブラを選んでいただきたいと思います。

0.2 フローチャート用紙とコーディング用紙

BASIC言語の普及により、コーディングとはキーボードから直接入力して行くもの、というのが一般的な常識になりつつあります。アセンブリ言語でも1画面におさまるくらい(20~25ステップ以内)のテスト・プログラムであれば、内蔵のアセンブラから直接打ち込んで良いかも知れませんが、それでも一応フローチャートなどを書いてからコーディングしてほしいものです。一度フローチャートを書くことによって頭の中で考えていたアルゴリズムにミスがなかったかどうかの確認もできますし、しっかりしたフローチャートが細部にわたって書いてあれば、かえってコーディングの必要がないくらいです。

そこで皆さんには、フローチャートを書くための用紙を用意しておいてほしいのですが、別に専用のものでなくても一般のグラフ用紙などでかまいません。筆者などは9インチ幅のプリンタ用紙をフローチャート用に愛用しているくらいです。

また、PC-9801の内蔵アセンブラを用いて、8086のソース・プログラムをコーディングするためのコーディング用紙も用意しておくくと便利ですが、これも普通の大学ノートやルーズ・リーフなどでB5版大(月刊誌の大きさ)のものが使いやすいようです。

下図にコーディング用紙の記入例を示します。

ADDR	LABEL	MNEM	OPERAND	COMMENTS
0000		MOV	AX, 0000H	
		MOV	DS, AX	
		XOR	BX, BX	
		MOV	AL, ECH	
		MOV	CX, 2000	
000C	NCHR	MOV	[BX], AL	
		INC	BX	
		INC	BX	
		LOOP	NCHR	
0012		HLT		
0013				

コーディング

フローチャート

フローチャート用紙

コーディング用紙

0・3 数の豆知識

コンピュータ関係の参考書や専門誌を読んでいると、よくビットとかバイトとかいう単語が目につきます。本書でも、説明なしにこれらの言葉を使用して来ました。皆さんにはすでにおなじみの単語だと思いますが、特にアセンブリ言語を理解するためには、これらの意味をよく知っておく必要がありますので、もう一度確認しておきたいと思います。

ビット

まず、ビット (BIT) というのは数の最小の単位で2進数の桁数に当たります。ですから、4ビットで表現できる最大の数は、

1 1 1 1

であり、16ビットで表現できる最大の数と言えば、

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

となります。これは10進数では65535に相当します。

バイト

このビットを8個集めたものがバイト (BYTE) です。すなわち、8ビットは1バイトであり、16ビットは2バイト、32ビットであれば4バイトと言い換えることができます。

ワード

ダブル・ワード

さらに、2バイトをワードと、4バイト (2ワード) をダブル・ワードと言う場合がありますので、覚えておくと便利です。

1ビット			
8ビット	1バイト		
16ビット	2バイト	1ワード	
32ビット	4バイト	2ワード	1ダブル・ワード

また、コンピュータでは、いろいろな表現の数字が使用されます。たとえば10進数の64であれば次のようにいろいろにあらわすことができます。

2進数 → 1 0 0 0 0 0 0 0

8進数 → 1 0 0

10進数 → 6 4

16進数 → 4 0

このまま単に数字だけを記述したのでは、何進数で表現されているのか区別することができません。そこでアセンブリ言語では次のように区別することが慣習になっています。

2進数 — 末尾にBをつける (BINARY)

8進数 — 末尾にQまたはOをつける (OCTAL)

10進数——末尾にDをつけるかそのまま (DECIMAL)

16進数——頭に\$をつけるか末尾にHをつける (HEXA)

先ほどの例 (10進数の64) にあてはめてみましょう。

2進数——10000000B

8進数——100Qまたは1000

10進数——64Dまたは64

16進数——\$40または40H

ところで、コンピュータは最終的に2進数を扱うのですが、人間は0と1のら列ではわかりにくいですから、普通は16進数に変換してあります。2進数を下位から4桁ずつに分けると、それぞれが0～15の数になります。その0～15の数に0～Fを割り当てて1桁としたものが16進数です。

最近8進数はあまり使いませんが、2進数を3桁ずつに分けて、それぞれを1桁にしたものが8進数です。

このように、2進数と16進数 (8進数) の変換は比較的容易ですが、16進数と10進数の変換を直接行なうのは意外にたいへんです。そのような場合にはBASICの命令 (HEXSなど) を用いるか、一度2進数を介して変換すると簡単です。

次の2ページに、1バイトで扱うことのできる数値を、2進数、16進数、10進数 (符号なし)、10進数 (符号あり) のそれぞれで表現した表を掲載します。この表は紙面の関係から2つのページに分かれています。下記のプログラムを走らせることによって、1つにまとめた変換表をプリンタ (PC-8822、8821、8023など) に出力することができます。いろいろな表現形式に親しんでください。

2進	8進	16進
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	10	8
1001	11	9
1010	12	A
1011	13	B
1100	14	C
1101	15	D
1110	16	E
1111	17	F

```
100 DEFINT A-Z
110 LPRINT CHR$(&H1B);'Q';CHR$(&H1B);'!';CHR$(&HC);
120 FOR K=0 TO 63
130   FOR I=K TO 255 STEP 64
140     LPRINT RIGHT$('0'+HEX$(I),2);'  ';
150     FOR J=7 TO 0 STEP -1
160       IF 0<(I AND (2^J)) THEN LPRINT '1'; ELSE LPRINT '0';
170     NEXT J
180     LPRINT USING ' ### ':I;
190     IF 127<I THEN LPRINT USING '####';I-256; ELSE LPRINT USING '####';I;
200     LPRINT SPACE$(8);
210   NEXT I
220   LPRINT
230 NEXT K
```



00	00000000	0	0	2A	00101010	42	42	54	01010100	84	84
01	00000001	1	1	2B	00101011	43	43	55	01010101	85	85
02	00000010	2	2	2C	00101100	44	44	56	01010110	86	86
03	00000011	3	3	2D	00101101	45	45	57	01010111	87	87
04	00000100	4	4	2E	00101110	46	46	58	01011000	88	88
05	00000101	5	5	2F	00101111	47	47	59	01011001	89	89
06	00000110	6	6	30	00110000	48	48	5A	01011010	90	90
07	00000111	7	7	31	00110001	49	49	5B	01011011	91	91
08	00001000	8	8	32	00110010	50	50	5C	01011100	92	92
09	00001001	9	9	33	00110011	51	51	5D	01011101	93	93
0A	00001010	10	10	34	00110100	52	52	5E	01011110	94	94
0B	00001011	11	11	35	00110101	53	53	5F	01011111	95	95
0C	00001100	12	12	36	00110110	54	54	60	01100000	96	96
0D	00001101	13	13	37	00110111	55	55	61	01100001	97	97
0E	00001110	14	14	38	00111000	56	56	62	01100010	98	98
0F	00001111	15	15	39	00111001	57	57	63	01100011	99	99
10	00010000	16	16	3A	00111010	58	58	64	01100100	100	100
11	00010001	17	17	3B	00111011	59	59	65	01100101	101	101
12	00010010	18	18	3C	00111100	60	60	66	01100110	102	102
13	00010011	19	19	3D	00111101	61	61	67	01100111	103	103
14	00010100	20	20	3E	00111110	62	62	68	01101000	104	104
15	00010101	21	21	3F	00111111	63	63	69	01101001	105	105
16	00010110	22	22	40	01000000	64	64	6A	01101010	106	106
17	00010111	23	23	41	01000001	65	65	6B	01101011	107	107
18	00011000	24	24	42	01000010	66	66	6C	01101100	108	108
19	00011001	25	25	43	01000011	67	67	6D	01101101	109	109
1A	00011010	26	26	44	01000100	68	68	6E	01101110	110	110
1B	00011011	27	27	45	01000101	69	69	6F	01101111	111	111
1C	00011100	28	28	46	01000110	70	70	70	01110000	112	112
1D	00011101	29	29	47	01000111	71	71	71	01110001	113	113
1E	00011110	30	30	48	01001000	72	72	72	01110010	114	114
1F	00011111	31	31	49	01001001	73	73	73	01110011	115	115
20	00100000	32	32	4A	01001010	74	74	74	01110100	116	116
21	00100001	33	33	4B	01001011	75	75	75	01110101	117	117
22	00100010	34	34	4C	01001100	76	76	76	01110110	118	118
23	00100011	35	35	4D	01001101	77	77	77	01110111	119	119
24	00100100	36	36	4E	01001110	78	78	78	01111000	120	120
25	00100101	37	37	4F	01001111	79	79	79	01111001	121	121
26	00100110	38	38	50	01010000	80	80	7A	01111010	122	122
27	00100111	39	39	51	01010001	81	81	7B	01111011	123	123
28	00101000	40	40	52	01010010	82	82	7C	01111100	124	124
29	00101001	41	41	53	01010011	83	83	7D	01111101	125	125
								7E	01111110	126	126

7F	01111111	127	127	AA	10101010	170	-86	D5	11010101	213	-43
80	10000000	128	-128	AB	10101011	171	-85	D6	11010110	214	-42
81	10000001	129	-127	AC	10101100	172	-84	D7	11010111	215	-41
82	10000010	130	-126	AD	10101101	173	-83	D8	11011000	216	-40
83	10000011	131	-125	AE	10101110	174	-82	D9	11011001	217	-39
84	10000100	132	-124	AF	10101111	175	-81	DA	11011010	218	-38
85	10000101	133	-123	B0	10110000	176	-80	DB	11011011	219	-37
86	10000110	134	-122	B1	10110001	177	-79	DC	11011100	220	-36
87	10000111	135	-121	B2	10110010	178	-78	DD	11011101	221	-35
88	10001000	136	-120	B3	10110011	179	-77	DE	11011110	222	-34
89	10001001	137	-119	B4	10110100	180	-76	DF	11011111	223	-33
8A	10001010	138	-118	B5	10110101	181	-75	E0	11100000	224	-32
8B	10001011	139	-117	B6	10110110	182	-74	E1	11100001	225	-31
8C	10001100	140	-116	B7	10110111	183	-73	E2	11100010	226	-30
8D	10001101	141	-115	B8	10111000	184	-72	E3	11100011	227	-29
8E	10001110	142	-114	B9	10111001	185	-71	E4	11100100	228	-28
8F	10001111	143	-113	BA	10111010	186	-70	E5	11100101	229	-27
90	10010000	144	-112	BB	10111011	187	-69	E6	11100110	230	-26
91	10010001	145	-111	BC	10111100	188	-68	E7	11100111	231	-25
92	10010010	146	-110	BD	10111101	189	-67	E8	11101000	232	-24
93	10010011	147	-109	BE	10111110	190	-66	E9	11101001	233	-23
94	10010100	148	-108	BF	10111111	191	-65	EA	11101010	234	-22
95	10010101	149	-107	C0	11000000	192	-64	EB	11101011	235	-21
96	10010110	150	-106	C1	11000001	193	-63	EC	11101100	236	-20
97	10010111	151	-105	C2	11000010	194	-62	ED	11101101	237	-19
98	10011000	152	-104	C3	11000011	195	-61	EE	11101110	238	-18
99	10011001	153	-103	C4	11000100	196	-60	EF	11101111	239	-17
9A	10011010	154	-102	C5	11000101	197	-59	F0	11110000	240	-16
9B	10011011	155	-101	C6	11000110	198	-58	F1	11110001	241	-15
9C	10011100	156	-100	C7	11000111	199	-57	F2	11110010	242	-14
9D	10011101	157	-99	C8	11001000	200	-56	F3	11110011	243	-13
9E	10011110	158	-98	C9	11001001	201	-55	F4	11110100	244	-12
9F	10011111	159	-97	CA	11001010	202	-54	F5	11110101	245	-11
A0	10100000	160	-96	CB	11001011	203	-53	F6	11110110	246	-10
A1	10100001	161	-95	CC	11001100	204	-52	F7	11110111	247	-9
A2	10100010	162	-94	CD	11001101	205	-51	F8	11111000	248	-8
A3	10100011	163	-93	CE	11001110	206	-50	F9	11111001	249	-7
A4	10100100	164	-92	CF	11001111	207	-49	FA	11111010	250	-6
A5	10100101	165	-91	D0	11010000	208	-48	FB	11111011	251	-5
A6	10100110	166	-90	D1	11010001	209	-47	FC	11111100	252	-4
A7	10100111	167	-89	D2	11010010	210	-46	FD	11111101	253	-3
A8	10101000	168	-88	D3	11010011	211	-45	FE	11111110	254	-2
A9	10101001	169	-87	D4	11010100	212	-44	FF	11111111	255	-1

1章 PC-9801の基礎知識

1.0 PC-9801のOS

すでにPC-9801を長い間使い込んできた皆さんは、ユーザーズ・マニュアルなどで、PC-9801の仕様を熟知していらっしゃるものと思います。

しかし、この項ではPC-9801を購入されて間がない皆さんのために、ソフトウェアの立場からPC-9801の基本仕様を述べておきたいと思います。

μPD8086

ご存知のように、PC-9801のCPUはμPD8086です。このCPUは1978年に米インテル社が生産を開始し、16ビットのマイクロ・プロセッサとしては最大のシェアを誇る8086と同等の機能(フル・コンパチブル)を持つマイクロ・プロセッサです。

この8086は、16ビットCPUとしての性能の良さはもちろんのこと、世界的なヒット作である8080と上位コンパチブルな命令群を有するため、8080用のアセンブリ言語で書かれたプログラムをほとんどそのまま流用できる点に大きな特色があります。このように、過去の長い間にわたって開発されて来たソフトウェアが利用できることはコンピュータにとっては非常に重要です。

CP/M-86 MS-DOS

さて、8086用のOS(オペレーティング・システム)としては、CP/M-86^{*1}やMS-DOS^{*2}が有名ですが、PC-9801でも、これらのOSがサポートされています(オプション)。

そしてPC-9801にROMとして標準搭載されているOS(言語)は、まぎれもないN88-BASIC(86)です。このため本書でのアセンブリ言語の学習もN88-BASIC(86)の下で行っているのです。

N88-BASIC(86)

N88-BASIC

このN88-BASIC(86)はマイクロソフト社がPC-8801のために開発したN88-BASICを、日本電気がPC-9801用につくり直したもので、BASICの命令は大部分がコンパチブルです。また、PC-8001用にBASIC言語で書かれたプログラムも利用できるように、N-BASIC(86)もサポートされています(オプション)。

N-BASIC(86)

以上の点から、PC-9801というパーソナル・コンピュータが、16ビット・マシンでありながら、PC-8001やPC-8801のために開発された多くの有用なソフトウェアを無駄にすることのない機種に仕上げられていることがわかります。

*1 CP/M-86はDIGITAL RESEARCH社のトレード・マークです。

*2 MS-DOSはMICROSOFT社のトレード・マークです。

1.1 N₈₈-BASICのキャラクタ

下図は、皆さん良くご存知のキャラクタ・コード表ですが、これらキャラクタ・コードの中で20H未満のものはコントロール・コードになっています。コントロール・コードの中で、良く使用されるものの意味を示します。

コード	シンボル	意味	コード	シンボル	意味
07	B _L	BELL	0B	H _M	HOME
08	B _S	BACK SPACE	0C	C _L	CLEAR
09	H _T	HORIZONTAL TAB	0D	C _R	CARRIAGE RETURN
0A	L _F	LINE FEED	7F	—	DELETE

08Hと7FHのコードは、同じように1文字消去を表わしますが、一般的なターミナルなどでは7FHを使っていることがほとんどです。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		D _E		0	@	P		p			一	タ	ミ	=	×	
1	S _H	D _I	!	I	A	Q	a	q			。	ア	チ	ム	円	
2	S _X	D ₂	"	2	B	R	b	r			「	イ	ツ	メ	年	
3	E _X	D ₃	#	3	C	S	c	s			」	ウ	テ	モ	月	
4	E _T	D ₄	\$	4	D	T	d	t			、	エ	ト	ヤ	日	
5	E _Q	N _K	%	5	E	U	e	u			・	オ	ナ	ユ	時	
6	A _K	S _N	&	6	F	V	f	v			ヲ	カ	ニ	ヨ	分	
7	B _L	E _B	/	7	G	W	g	w			ア	キ	ヌ	ラ	秒	
8	B _S	C _N	(	8	H	X	h	x			イ	ク	ネ	リ	♠	
9	H _T	E _M	)	9	I	Y	i	y			ウ	ケ	ノ	ル	♥	
A	L _F	S _B	*	:	J	Z	j	z			エ	コ	ハ	レ	♦	
B	H _M	E _C	+	;	K	[	k	{			オ	サ	ヒ	ロ	♣	
C	C _L	→	,	<	L	¥	l				ヤ	シ	フ	ワ	●	
D	C _R	←	-	=	M	]	m	}			ユ	ス	ヘ	ン	○	
E	S ₀	↑	.	>	N	^	n	~			ヨ	セ	ホ	.	◁	
F	S _I	↓	/	?	O	_	o				ツ	ソ	マ	°	▷	

1.2 PC-9801のメモリ・マップ

メモリ・マップ

バンク切り換え

日本電気のパーソナル・コンピュータ、PCシリーズも、PC-8001からPC-6001、PC-8801へと拡張されるにつれて、そのメモリ・マップも複雑化して来ました。これは、今までの機種がCPUにZ80(μPD780)を採用していたためで、Z80で利用できる64Kバイトの限られた空間に、より多くの機能をつめ込むためにメーカーが苦戦した結果だと言えます。特にPC-8801などは並列に並べたいくつものメモリ空間(バンク)を切り換えて使用するバンク切り換えの手法を大幅に採用したため、メモリ・マップの複雑化は極みに達し、私達ユーザには容易に扱えないほどです。

これに反して、PC-9801のメモリ・マップは非常にすっきりした構造をしています。次ページにPC-9801のメモリ・マップを示します。

アクセス

従来の8ビットCPUのアドレスが4桁の16進数で表現されたのに対して、8086(μPD8086)のアドレスは5桁の16進数(00000H~FFFFFFH)で表現できます。すなわち、直接アクセスすることのできるメモリ空間が一挙に16倍になり、このためメモリ・マップの簡略化(一本化)が可能になったわけです。

なお、次ページのメモリ・マップは非常に簡略化した形となっており、実際にはマップ中のそれぞれの項目がさらに細分されたメモリ・マップが用意されています。詳しくはPC-9801購入時に付属される「PC-9801ユーザーズ・マニュアル」19-47から19-52に説明がありますので、興味がおありの方はご参照ください。

本書での機械語エリア

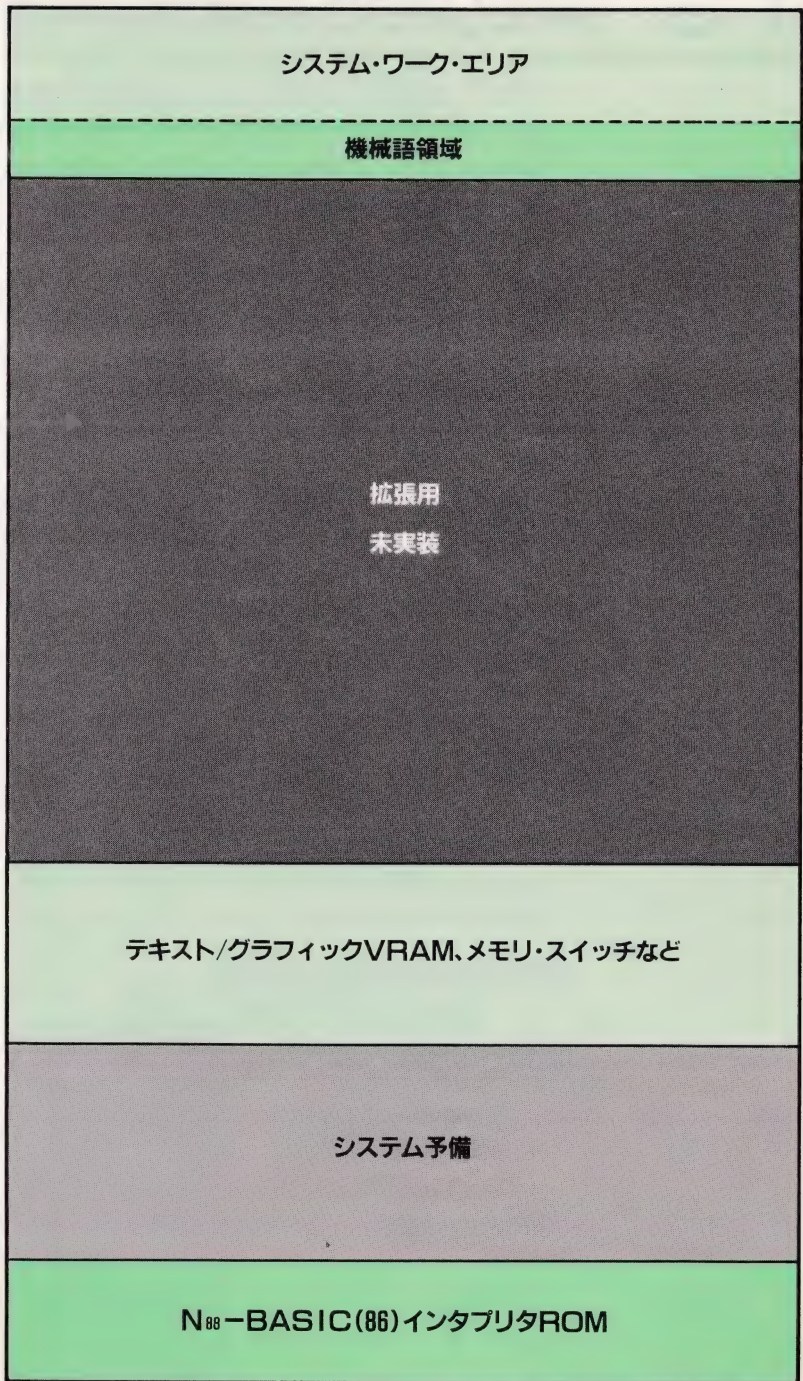
さて、皆さんがアセンブリ言語を学習して行くためには、機械語のプログラムを収納するための場所をメモリ空間の中でさがさなければなりません。とりあえず本書では、18000H番地から1FFFFH番地の間を機械語のために使用したいと思います。32Kバイトもありますので十分過ぎるくらいでしょう(なんと、PC-8001のN-BASICインタプリタでさえ24Kバイトです)。

CLEAR

標準のPC-9801では、00000H~1FFFFH番地にRAMが実装されており、システムではこの領域をワーク・エリアやN88-BASIC(86)のテキストを収めるために使用しますが、上位(1FFFFHに近い方)から任意の量のメモリを機械語領域として確保するための命令(CLEAR)が用意されています。もちろん、あまり多くのメモリを機械語領域として確保することはできません。

アドレス
↓
00000H
10000H
20000H
30000H
40000H
50000H
60000H
70000H
80000H
90000H
A0000H
B0000H
C0000H
D0000H
E0000H
F0000H
FFFFFH

メモリ・マップ



1.3 メモリのセグメンテーション

B

●2進数の末尾には普通アルファベットのBをつける(BINARYの略)

PC-9801が、00000H~FFFFFH番地(1Mバイト)のメモリ空間を有することは「メモリ・マップ」の項で述べました。したがって、アドレスも20ビット(0000 0000 0000 000 0 0000B~1111 1111 1111 1111 1111B)でなければ表現できないことは明白です。

しかし、8086(μPD8086)は16ビットのCPUとして設計されているために、扱うことのできるアドレスも16ビットに限定されています。そこで、16ビットのアドレス指定によって20ビットのアドレス空間をアクセスするための特別なメカニズムが8086の特色として用意されているのですが、このメカニズムが、これから説明するセグメントです。

セグメント

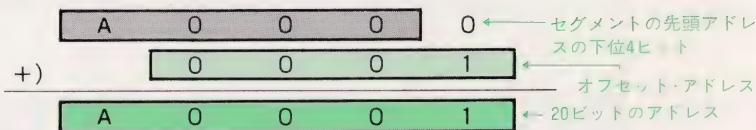
オフセット・アドレス

つまり、1Mバイトのメモリ空間を任意の個数の領域(セグメント)に分けた上で、そのセグメント中の論理的な(理論上の)アドレス(オフセット)によって間接的にアドレス指定を行なうことになります。この場合、1つのセグメント中のアドレスは0000H~FFFFFH番地(16ビット)で指定できる64Kバイトになっています。

説明が少し抽象的でややこしくなっていました。いくつか具体例をあげて考え方を整理しなおしてみましょう。

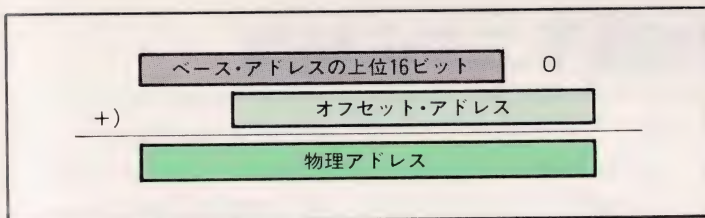
いま、A0001H番地を指定することを考えてみます。

この場合には、A0000H番地からAFFFFFH番地までの64Kバイトを1つのセグメントと考えれば、オフセットのアドレスは0001H番地となります。



ベース・アドレス

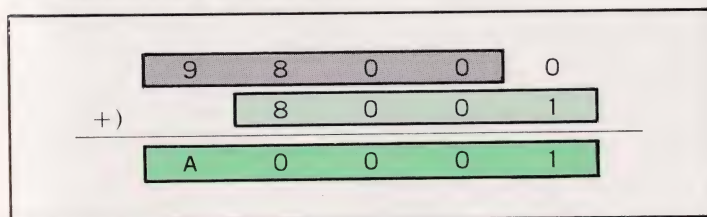
この図でもわかるように、セグメントの先頭アドレス(ベース・アドレスと言う)の下位4ビットが必ず0であることが決まれば、2つの16ビットの数値、すなわち、セグメント・ベース・アドレスの上位16ビットとオフセット・アドレスによって、20ビットのアドレスを構成することができ、00000H~FFFFFH番地(1Mバイト)全てを指定することが可能なわけです。



8086がメモリをアクセスする際には、セグメントのベース・アドレスとして設定してある16ビットの値の末尾4ビットに0を加え（実際には4ビット分左にシフトする）さらにオフセット・アドレスを加えたものを実際のアドレス（物理アドレス）としてアドレス指定を行います。

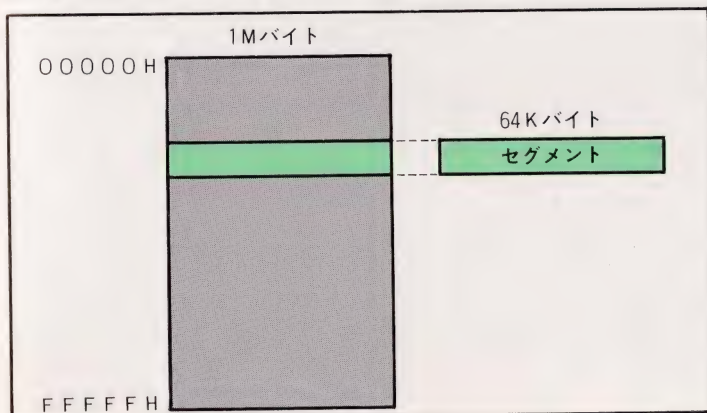
ベース・アドレスとしては、ユーザが任意に設定することができますが、普通は、0000H（実際には00000H番地がセグメント・ベースとなる）、1000H、2000H……、のように切りの良い値や、1800H（実際には18000H番地がセグメント・ベースとなる）のような値に設定することが多いようです。

もう1つ例をあげますと、ベース・アドレスとして9800H（実際には98000H番地がベース・アドレスとなる）を設定してある場合に、先ほどと同じA0001H番地を指定するためのオフセット・アドレスは8001H番地になります。



次図に、メモリのセグメンテーションを示します。

セグメンテーション



1・4 機械語モニタの概要

一般にパーソナル・コンピュータ用の機械語モニタは簡単なものが多く、最近の低価格マシン（PC-6001など）にはモニタが内蔵されていない機種さえ見うけられます。一体、機械語モニタのないパーソナル・コンピュータで機械語の勉強ができるのでしょうか？ 私には疑問です。

機械語モニタ

幸いPC-9801の機械語モニタは、PC-8801の多機能モニタの機能を全て受け継ぎ、大変充実しています。アセンブルのためのAコマンドと逆アセンブルのためのLコマンドがディスク・ベースでないといえないのが、欠点と言えば欠点ですが、これだけで多くの機能を

コマンド	意味	機能
A	アセンブル	入力した1行をアセンブルします
B	ベース	数値の表現形式を変更します（8進↔16進）
C	チェンジ・セグメント	セグメントのベース・アドレスを設定します
D	ダンプ・メモリ	メモリの内容をダンプします
E	エディット・メモリ	メモリの内容をスクリーン・エディットします
F	フィル・メモリ	メモリの内容を定数で埋めます
G	ゴー	ユーザのプログラムを実行します
I	インプット・ポート	入力ポートの値を読み込みます
L	ディス・アセンブル	機械語を逆アセンブルします
M	ムーブ・メモリ	メモリの内容をブロック転送します
O	アウトプット・ポート	出力ポートへデータを出力します
P	プリンタ・スイッチ	プリンタへの出力をコントロールします
R	リード・テープ	テープからデータをロードします
S	セット・メモリ	メモリにデータをセットします
TM	テスト・メモリ	メモリをテストします
V	ベリファイ・テープ	テープの内容とメモリの内容を比較します
W	ライト・テープ	メモリの内容をテープにセーブします
X	イグザミン・レジスタ	8086のレジスタの値を調べ変更します
CTRL-A	HELP	モニタのコマンドおよびパラメータを表示します
CTRL-B	リターン・ベーシック	BASICモードへ復帰します
CTRL-D	ダンプ・ディスク	ディスクの内容をダンプします
CTRL-R	リード・ディスク	ディスクからデータをロードします
CTRL-W	ライト・ディスク	ディスクへデータをセーブします

備えていれば、アセンブリ言語の学習も非常に行ないやすいと思われます。

ところで、23種あるコマンドの中に、新しくPC-8801にはなかったC(チェンジ・セグメント)コマンドが増設されています。これは、機械語モニタの各コマンドで参照するセグメントのベース・アドレスを設定するためのコマンドで、Cコマンド以外の全てのコマンドのアドレス指定はメモリ・マップ上の物理アドレスではなく、オフセット・アドレスで行ないます。たとえば、Cコマンドによってベース・アドレスをA0000H番地に設定(CA000 ☒)した後、Dコマンドで1000H番地を指定(D1000 ☒)した場合、実際にダンプされるのはA1000H番地からの内容となります。

次に、**HELP** キーを押すことによって表示される、コマンドおよびパラメータの一覧表を示します。

Monitor has following commands.

a[<address>]	assemble source text lines.
bh or bq	select radix (hexa decimal or octal).
c[<address>]	change segment base.
d[<start>][C,<end>]	dump contents of memory.
e[<start>]	change contents of memory by screen editor.
f[<start>,<end>,<const>]	fill memory by the constant.
g[<start>][C,<break1>][C,<break2>]	execute user program.
i[<port>]	read input port.
l[<start>][C,<end>]	dis-assemble program in memory.
m[<start:s>,<end:s>,<start:d>]	move contents of memory block(s=src,d=dest:)
o[<port>,<new value>]	output new value to the port.
p	toggle print switch.
r[1:2:][C[<file name>]]	read cassette file.
s[<address>] or s[<switch name>]	substitute memory contents.
tm	test memory.
v[1:2:][C[<file name>]]	verify cassette file.
w[1:2:][C[<file name>]]	write cassette file.
	,<start>,<end>
x or x [<register name>]	dump CPU registers or change the registers.
CNTL-b	return to BASIC.
CNTL-d [<dr>[C,<sur>],<tr>,<sec>]	dump contents of sectors.
	[C,<tr>,<sec>]
CNTL-r [<dr>[C,<sur>],<tr>,<sec>]	read sectors into memory.
	,<start>,<end>
CNTL-w [<dr>[C,<sur>],<tr>,<sec>]	write contents of memory into sectors.
	,<start>,<end>

1.5 機械語モニタの起動方法

アセンブリ言語を扱うためには、N88-BASIC(86)のコマンド待ちの状態からMONコマンドを入力して、機械語モニタを起動しなくてはなりませんので、その手順を説明します。

まず、PC-9801本体と周辺機器（とりあえずCRTモニタとディスク・ユニット）を接続します。正常に接続してあることを確認したら、各ディスク・ユニットに電源を投入し、システム・ディスクをセットします。

ここで、PC-9801本体に電源を投入すると数秒して次のメッセージが表示されます。

```
Disk version
How many files(0-15)? █
```

↑ カーソル

普通ここでは、使用するファイルの数を入力するのですが、リターンキーのみを入力しておいてください。

すると、次のメッセージが表示されます（表示される数字は異なる場合があります）。

```
Disk version
How many files(0-15)?
NEC N-88 BASIC(86) version 1.0
92082 Bytes free
Ok
█
```

これで、N88-BASIC(86)のDISK VERSIONが起動し、いつでもBASICのコマンドを受け付ける体制がととのったわけです。

ここで、MONコマンドを入力すればいつでも機械語モニタを起動することができますが、その前に機械語を収めるためのメモリ領域を指定しておきましょう。

先ほど決めたように、18000H番地から1FFFFH番地まで(32Kバイト)を機械語のための領域として指定し、確保しておきます。これらの指定は一度行なっておけば、以後はMONコマンドを入力するのみで機械語モニタを起動しなおすことができます。

とりあえず次のように入力しておいてください。

```

Disk version
How many files(0-15)?
NEC N-88 BASIC(86) version 1.0
92082 Bytes free
Ok
clear ,&h1800 ← 機械語領域を確保
Ok
def seg=&h1800 ← ベース・アドレスの指定
Ok
mon ← 機械語モニタの起動
hJ■

```

最初に入力する「CLEAR , &H1800」は18000H番地から後に機械語のプログラムを格納することを示し、N88-BASIC(86)が18000H番地以降を勝手に使用してしまうことを防ぎます。

次の「DEF SEG=&H1800」は、使用するセグメントのベース・アドレスを18000H番地に指定するために入力しておきます。これより以後、メモリを扱うための命令や関数（PEEK、POKE、MON）は全て18000H番地をベース・アドレスとして実行されます。たとえば、次のように入力すると18005H番地の内容が表示されます。

```
print peek(5)
```

逆に、機械語モニタからBASICのコマンド待ちにもどるためには、**CTRL** キーを押しながら **B** キーを押してください。「OK」が表示されて、N88-BASIC(86)にもどります。

```

Disk version
How many files(0-15)?
NEC N-88 BASIC(86) version 1.0
92082 Bytes free
Ok
clear ,&h1800
Ok
def seg=&h1800
Ok
mon
hJ^B
Ok
■

```

CLEAR

DEF SEG

2章 8086の基礎知識

2・0 8086の生い立ち

ここで、8086の生い立ちについて少し話しておきましょう。

4004

世界で一番最初に開発されたCPU(マイクロ・プロセッサ)は、4ビットの4004(i4004)です。4004は、インテル社が電卓用に設計し、1971年12月に発表されましたが、この史上初のCPUが、日本の中小企業であったビジコン社によって発案された事実も有名です。

8008

次にインテル社は、コンピュータの端末用として、8ビットの8008(i8008)を開発しました。

しかし、上記の4004および8008は、いずれも専用機を意図していたため汎用性に乏しく、一部の技術者やマニアが知ったのみで、広く取り上げられるまでには至りませんでした。

8080

6800

この後、各半導体のメーカーは、より汎用化し能力を強化したCPUの開発に着手し、1974年には先のインテル社が8080(i8080)を、モトローラ社が6800(M6800)を発表したことにより、マイクロ・プロセッサというものがコンピュータ業界で注目を受けはじめました。

インテル社の8080は、8008の改良版であったにもかかわらず、それまでの専用機から脱却し、急速に8ビット・マイクロ・プロセッサの標準機としての立場を築いてしまいました。現在では多くの半導体メーカーが、8080のセカンド・ソースとなっています。

Z80

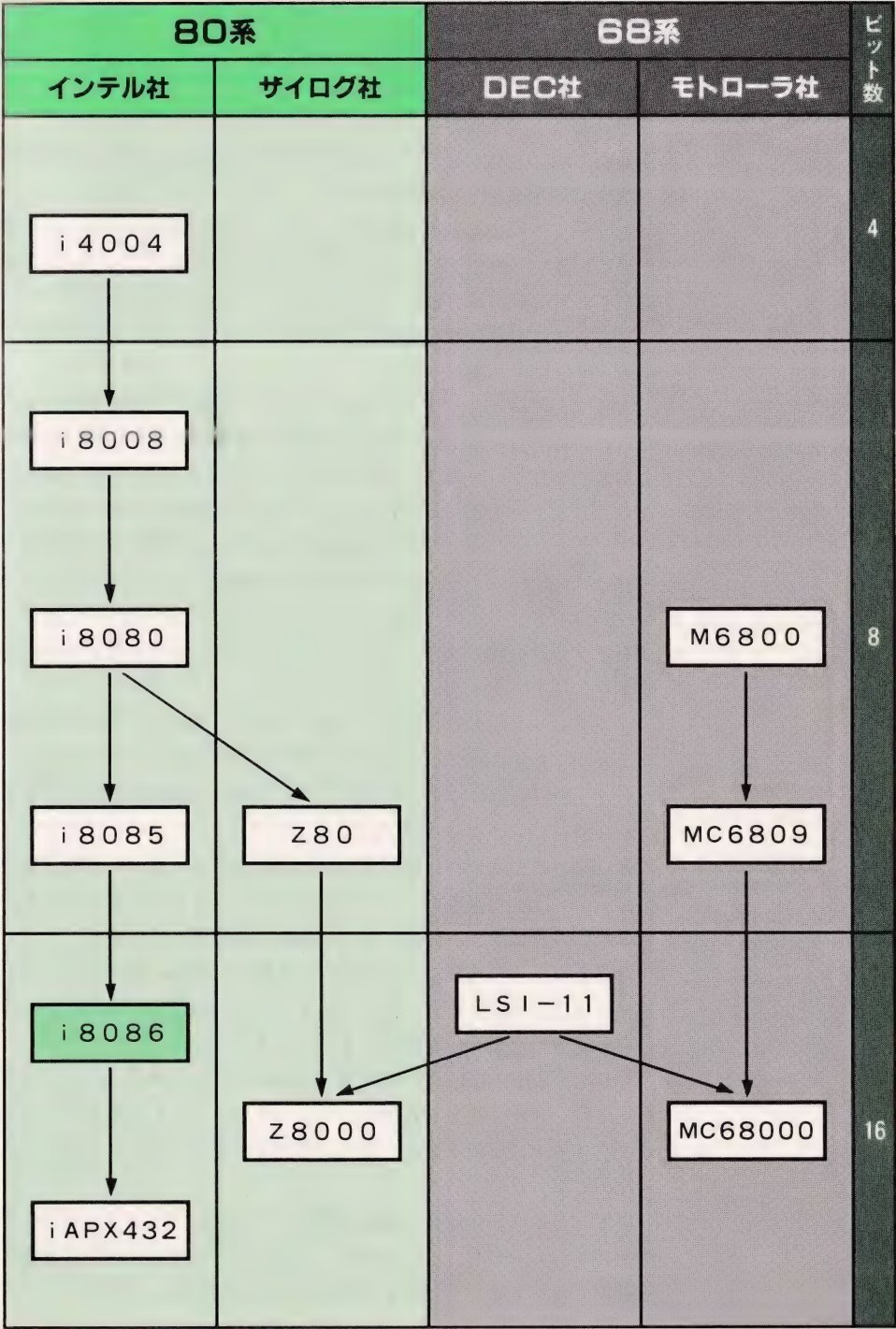
ところで、8080は日本人である嶋正利氏によって設計されましたが、同氏がザイログ社に移って設計したのがZ80です。Z80は、8080と完全な上位互換性(コンパチビリティ)を持っていたためもある、圧倒的なシェアを獲得し、日本でも主要なパーソナル・コンピュータのほとんどがZ80を搭載していることは皆さんも良く御存知だと思います。

8085

6809

Z80の他にも、インテル社自身が8080を改良した8085(i8085)を発表していますし、モトローラ社も6800の流れを継いで、究極の8ビット・マイクロ・プロセッサとさえ呼ばれる6809(MC6809)を発表し、8ビットの時代は隆盛を迎えました。

しかし、これら8ビットのマイクロ・プロセッサに対する要求が高まるにつれて、能力を超えるものが後をたちません。次に列記するものは、8ビットのマイクロ・プロセッサに対する不満のほんの一例です。ついこの間まで、メモリ容量が多いとか、速度が速いなどと言って喜んでいたのが嘘のようです。



- メモリ容量が少ない
- 処理速度（実行速度）が遅い
- 演算機能が乏しい（乗除算命令がない）
- アドレス指定の方法（アドレッシング・モード）が少ない
- 他のCPUと協調する（マルチ・プロセッサ）機能が欠けている
- 文字列を扱うための命令がない

そして、これらの不満を解消するため、STEPHEN P. MORSE氏、BRUCE RAVENEL氏等によって設計され、1978年にインテル社より発表されたのが、8086（i8086=iAPX86）です。

このような8086の生まれた背景は、他の16ビット・マイクロ・プロセッサとは全く異なります。

8086の発表後、ザイログ社、モトローラ社よりそれぞれ、Z8000、68000（MC68000）が発表されましたが、これらの背景となっているのは、自社の8ビット・マイクロ・プロセッサではなく、LSI-11です。LSI-11は、DEC社のミニ・コンピュータであるPDP-11の持つ機能を、同社およびウエスタンデジタル社の共同開発によって4チップのLSIに移植したものです。

したがって、Z8000および68000は、8ビット・マイクロ・プロセッサの拡張ではなくミニコンの1チップ化から生まれたもののものです（68系CPUの特色）。

コンピュータの世界に限らず、現在一流と言われているメーカーが作り出した製品には多少のばらつきがあるとは言え、使い物にならないような製品はまずありません。そこで評価の時重要になってくるのが「慣れ」という要素です。

長い間にわたって使い慣れたものほど使い勝手の良いものはありません。もし、私に複数のパーソナル・コンピュータを持つことが許されなければ、迷わず、一番使い慣れた機種を選択するでしょう。

たとえば、8086と68000を比較した場合、単に性能だけを比べれば68000の方が勝っていることは明白です。Z80と6809を比較した場合にも同じことが言えました。

しかし、Z80のシェアが8ビット市場の大半を占めたのは、8080と上位レベルで完全な互換性（コンパチビリティ）が得られたためです。ユーザは機能を追求した6809よりも使い慣れたZ80を選んだのです。

ほとんどのユーザが長い間慣れ親しんできた8ビットのマイクロ・プロセッサから派生した8086が、他の16ビット機を大きく引き離し短期間で名声を勝ち得たのも、当然のことと言えましょう。

8086

Z8000

68000

LSI-11

2・1 8086の設計思想

8086は、8ビットのCPUである8080(8085も含む)の欠点や限界を克服するために開発されたマイクロ・プロセッサです。すなわち、8086の長所を知りたいければ、8080の欠点を調べれば良いことになります。

マイクロ・プロセッサに対する評価が高まるにつれて、8080もしだいに大きなシステムで用いられるようになってきました。その結果8080の持つ64Kバイトのメモリ空間ではまかない切れなことが多くなってきたのです。これに対して8086には1Mバイトという桁違いのメモリ空間が用意されています。

また、現在CPUに任せたい仕事の中に8ビットでまかなえるデータが非常に少ないことも問題でした。8080では8ビットより長いデータは8ビットごとに分割して処理しなければならず、これは結果的な実行速度の低下を意味します。これに対して8086は16ビットのCPUですから16ビット(場合によっては32ビット)のデータを扱えるのはもちろんのこと、従来と同じように8ビットのデータを処理する機能もそのまま残されているため、短いデータに対しても効率良く処理が行なえますし、8080のために蓄積された多くのソフトウェアをそのまま走らせることができるのも大きな魅力です。

さらに、8080では演算機能が不足していましたが、8086には乗除算命令をはじめとして多くの命令が拡張されました。

以上のような8086の根本に流れている設計思想は、常にアセンブラではなく、高級言語への対応を十分に意識したものです。これは、8086を設計したインテル社においてアセンブラよりも早く開発されたプログラミング言語がPL/M(PL/M-86)であることから推測できます(PL/M-86自体はハンド・アSEMBルによってつくったのでしょうか!)

すなわち、高級言語で書かれたプログラムをコンパイラによって機械語に変換するような場合、より効率良く行なえることが8086の設計目標だったわけです。

読者の皆さんも、本書を読み進んで行くうちに、自然に8086の設計思想がわかってくると思います。幸いPC-9801の機械語モニタにはアSEMBルを行なうためのコマンドが用意されていますが、複雑な命令形式を数多く備えた8086で、今までの8ビットCPUの頃のようにハンド・アSEMBルを行なうのには問題がありそうです。

PL/M

●インテル社のマイクロ・コンピュータのために開発した高レベルの言語で、汎用コンパイラPL/Mのサブ・セットに近い形をとっている。PROGRAMMING LANGUAGE FOR MICROCOMPUTERの略。

2.2 8086のレジスタ・セット

レジスタ

CPUは、命令を実行したり各種の演算を行ったりするためのデータや情報を一時記憶しておくための場所として、いくつかのレジスタを持っているのが普通です。CPUのレジスタの概念を説明するために、よくBASIC等の変数が引き合いに出されますが、本当は両者は全く異なったものです（レジスタはハードウェアで構成されており、変数はソフトウェアによって実現している）。しかし、アセンブリ言語でいうレジスタがBASICの変数とほとんど同じ働きを持っていることも事実であるため、慣れないうちは「レジスタ≒変数」というように考えていてもかまわないと思います。

このように述べると、メモリもデータを記憶するための場所ですから、「レジスタ＝メモリ」と考えても良いように思えますが、まさにそのとおりです。ただ、メモリとレジスタの異なる点は、メモリがCPUの外部にあるのに対してレジスタはCPUに内蔵されていることで、万一レジスタが存在しない場合には演算命令などを実行するたびにCPUとメモリの間でデータのやり取りを行なわなければなりません。しかし、メモリのアクセスには時間がかかり、そのためにプログラムが長くなる（使用するメモリ容量が増加する）場合も多いため、頻繁に利用するデータを一時的に入れておける手近な場所としてレジスタが用意されているのです。

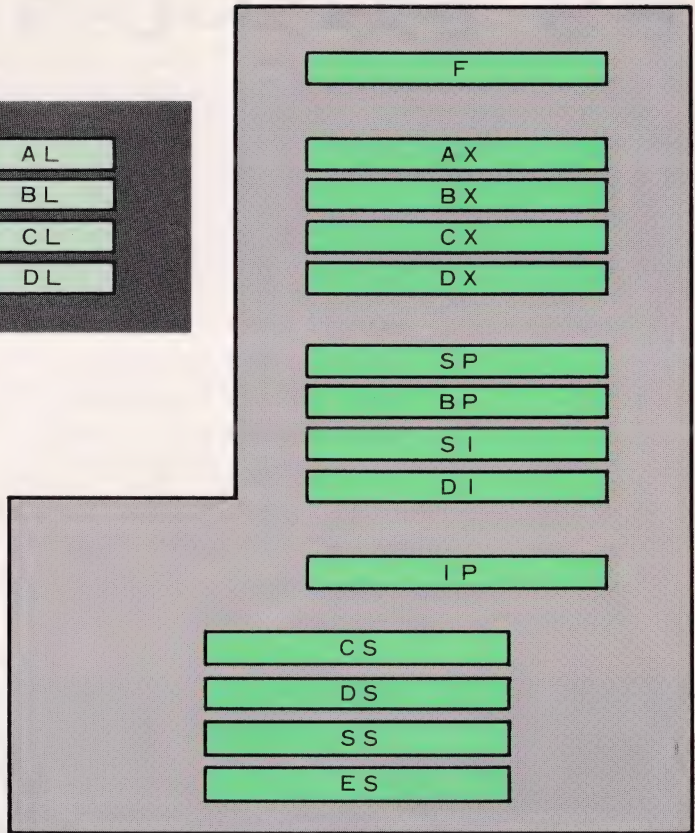
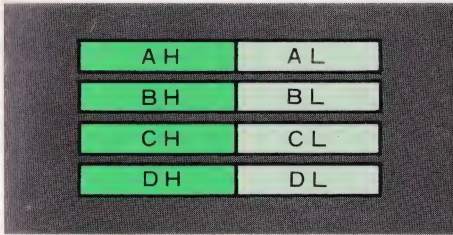
次ページは、8086（μPD8086）の持つレジスタを表にしたものです。

表からもわかるように、8086は16ビットのレジスタ14個を持っていますが、これらのレジスタは用途によっていくつかに分類することができます。

フラグ	F
汎用レジスタ	AX、BX、CX、DX
ポインタ／インデックス・レジスタ	SP、BP、SI、DI
命令ポインタ	IP
セグメント・レジスタ	CS、DS、SS、ES

また、汎用レジスタすなわちレジスタAX、レジスタBX、レジスタCX、レジスタDXは、それぞれ上下8ビットずつに分けて8ビットのレジスタとして使用することができます。たとえば、レジスタDXであれば、上位8ビットをレジスタDHとし、下位8ビットをレジスタDLとして、2個の8ビット・レジスタとして使用することもできます。

8086の内部レジスタ・セット



F	——	フラグ
AX	——	アキュムレータ
BX	——	ベース
CX	——	カウンタ
DX	——	データ
SP	——	スタック・ポインタ
BP	——	ベース・ポインタ
SI	——	ソース・インデクス・レジスタ
DI	——	デストネーション・インデクス・レジスタ
IP	——	命令ポインタ
CS	——	コード・セグメント・レジスタ
DS	——	データ・セグメント・レジスタ
SS	——	スタック・セグメント・レジスタ
ES	——	エクストラ・セグメント・レジスタ

2.3 セグメント・レジスタの役割

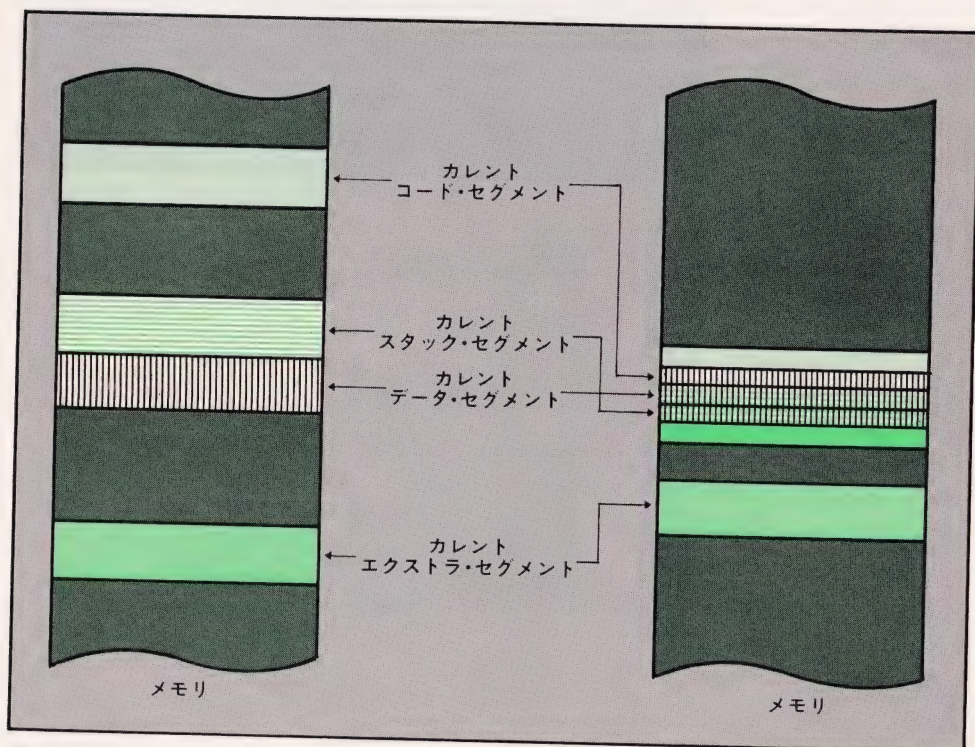
8086が、1Mバイトのメモリ空間を、複数の64Kバイトの領域(セグメント)に分割して利用することは「セグメンテーション」の項で説明しました。

ここで、「複数のセグメント」と述べましたが、実際に8086が同時に使用できるのは4つのセグメントで、それぞれに次の名称がついています。

カレント(現在の)

- ① カレント・コード・セグメント
- ② カレント・データ・セグメント
- ③ カレント・スタック・セグメント
- ④ カレント・エクストラ・セグメント

これら4つのセグメントは、00000H~FFFFFFH番地のどのアドレスから始まってもかまいませんし、下図の左のように完全に独立していても、右のように重なっていてもかまいません。場合によっては4つのセグメントのアドレスが完全に一致していることもあります。



8086の4個のセグメント・レジスタは、4つのカレント・セグメントのそれぞれが始まっているアドレス（ベース・アドレス）を指定するためのレジスタです。

コード・セグメント・レジスタ	CS	0
データ・セグメント・レジスタ	DS	0
スタック・セグメント・レジスタ	SS	0
エクストラ・セグメント・レジスタ	ES	0

すなわち、コード・セグメント・レジスタ（レジスタCS）に18000Hが入っている場合にはカレント・コード・セグメントは18000H番地から始まって27FFFH番地まで、データ・セグメント・レジスタ（レジスタDS）にA000Hが入っている場合にはカレント・データ・セグメントはA000H～AFFFFH番地です。

⑤コード・セグメント

主として、機械語のプログラムを置いておくためのセグメントです。CPUが実行するための命令をメモリ上から持ってくる（取り出す）ことを、命令のフェッチ（FETCHES）と言いますが、8086では、フェッチする命令は必ずコード・セグメント上になければいけません。

命令のフェッチ

また、場合によっては、コード・セグメントの一部をデータ・エリアとして使用することもあります（本書では多用）。

⑥データ・セグメント

8086がメモリをアクセス（読み出しおよび書き込み）する場合に、上記のフェッチと一部の例外を除いたほとんどのメモリ・アクセスが、このデータ・セグメントに対して行なわれます。例外に関しては後で詳述しますが、ポインタ・レジスタ（SPまたはBP）によってオフセット・アドレスを指定する場合にはスタック・セグメントに対してアクセスが行なわれ、文字列を扱うためのストリング命令ではエクストラ・セグメントに対してアクセスが行なわれる場合があります。

⑦スタック・セグメント

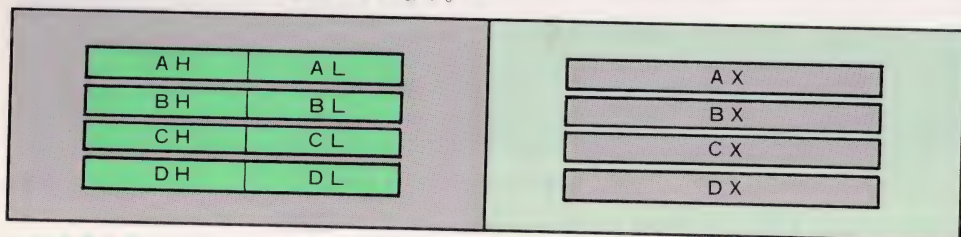
ポインタ・レジスタ（SPまたはBP）のオフセット・アドレスによってメモリ・アクセスを行なうのがスタック・セグメントです。

⑧エクストラ・セグメント

別々のセグメントに渡る（64Kバイトにおさまらない）2種類以上のデータを同時に扱うような場合、一方をデータ・セグメントとし、他方をエクストラ・セグメントとして扱うと便利です。

2・4 汎用レジスタの役割

8086は、AX、BX、CX、DXと、4個の16ビット汎用レジスタを持っています。



レジスタAX

レジスタBX

レジスタCX

レジスタDX

本書はこれら4個のレジスタを、それぞれ、レジスタAX、レジスタBX、レジスタCX、レジスタDXと呼んで行きます。また、それぞれのレジスタは、8ビットの2個のレジスタとして使用することもあります。この場合、たとえばレジスタAXを2個のレジスタとして使用する場合、レジスタAHとレジスタALとなります。

レジスタの名称は、アルファベットの先頭から、A、B、C、Dとつけたように見えますが、次のようにそれぞれ意味のある名称がつけてあります（後からこじつけた感もありますが）。

AX — アキュムレータ (ACCUMULATOR)

BX — ベース (BASE)

CX — カウント (COUNT)

DX — データ (DATA)

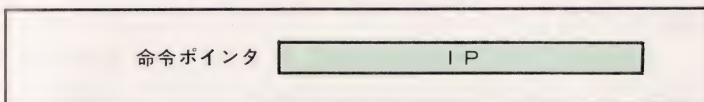
ほとんどの命令では、汎用レジスタ全てが平等の立場にあるため、どのレジスタを使用するかをユーザ（プログラマ）が勝手に決めることができます。たとえば、レジスタAXとレジスタBXの和を求める命令があれば同様に、レジスタBXとレジスタCXの和やレジスタDXとレジスタAXの和なども求める命令が存在し、さらに8ビットのレジスタ間、レジスタAHとレジスタDLの和やレジスタCHとレジスタCLの和などを求めることができます。これはほんの一例ですが、全ての汎用レジスタを組み合わせた場合、加算だけでも、いかに膨大な数の命令が使用できるか予想がつくと思います。

しかし、意味のある名称がつけてある以上、特定のレジスタを決められた目的のみに使用する命令もあります。詳しくはそれぞれの命令が出て来た時点で説明して行きますが、1つの例をあげておきますと、8086のストリング命令（文字列を操作する命令）において文字数を数える（カウントする）のに必ずレジスタCXを使うので、「COUNT」の「C」を取って「CX」の名称がついたのです。

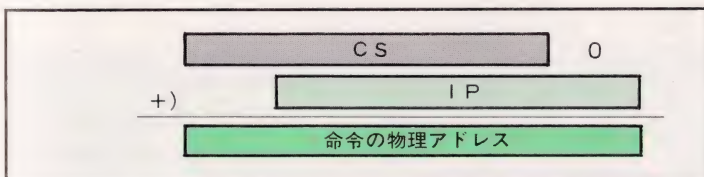
2.5 命令ポインタの役割

命令のフェッチは、すべてカレント・コード・セグメントに対して行なわれます。そこで、次に実行すべき命令の、コード・セグメントにおけるオフセット・アドレスを入れておくためのレジスタが必要になりますが、これが命令ポインタ (IP) です。

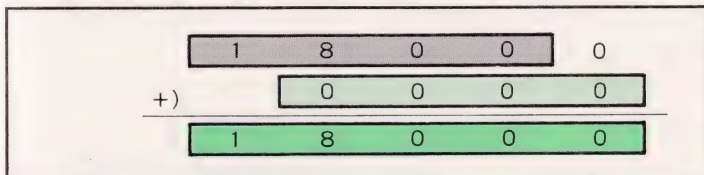
命令ポインタ



すなわち、フェッチを行なう物理アドレス (実際のアドレス) は、コード・セグメント・レジスタ (レジスタCS) と命令ポインタ (IP) を加えた値になります。



たとえば、コード・セグメント・レジスタ (CS) に1800Hが入っている場合に、命令ポインタに0000Hが入っていれば、8086は18000H番地に書かれた命令を実行します。命令がフェッチされると命令ポインタ (IP) の値が自動的にふやされ、次のフェッチに備えます。



他のレジスタと異なり、この命令ポインタ (IP) には直接データを代入するための命令が存在しません。なぜならば、サブルーチンをコールしたり、ジャンプすることが、命令ポインタ (IP) にデータを代入することに他ならないからです。

逆に言えば、命令ポインタ (IP) およびコード・セグメント・レジスタ (CS) の内容を変更する場合には十分に注意しなくてはなりません。どちらかのレジスタが不適当な値になった場合でも、8086は忠実に命令をフェッチし、全く意味のない命令を実行して行ってしまう。この状態が、一般的に「暴走」と言われるものです。

暴走

2・6 ポインタとインデクス・レジスタの役割

一般にメモリ上のアドレスを指定する方法として、直接指定と間接指定が考えられます。

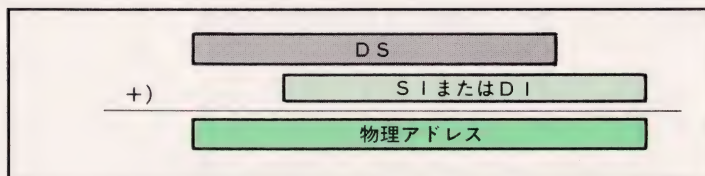
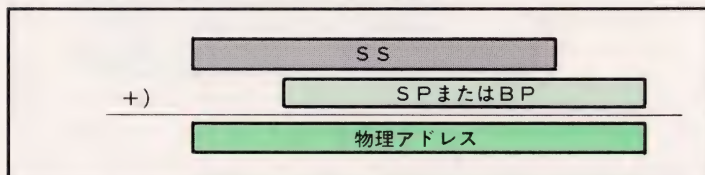
アドレスの直接指定
間接指定

前者の直接指定では、「5000H番地」とか「0005H番地」などのように直接にオフセット・アドレスを指定します。これに対して間接指定では、あらかじめ16ビットのレジスタ（たとえばレジスタBX）に、オフセット・アドレスを納めておき、「レジスタBXで指定するアドレス」とか「レジスタBXで指定する次のアドレス」などのように間接的にアドレスを指定します。

アドレスの間接指定には、上記のようにレジスタBXを用いることもできますが、8086には、特にアドレス指定を重点に行なうための4個のレジスタが用意されています。

スタック・ポインタ	SP
ベース・ポインタ	BP
ソース・インデクス・レジスタ	SI
デストネーション・インデクス・レジスタ	DI

これらのレジスタは、ポインタ（SPおよびBP）と、インデクス・レジスタ（SIおよびDI）に分類することができますが、普通インデクス・レジスタがカレント・データ・セグメントに対するオフセット・アドレスを示すのに反して、ポインタはカレント・スタック・セグメントに対するオフセット・アドレスを示します。



また8086では、これらのポインタおよびインデクス・レジスタを通常の汎用レジスタと同じような役割（たとえば演算）で使用することができます。

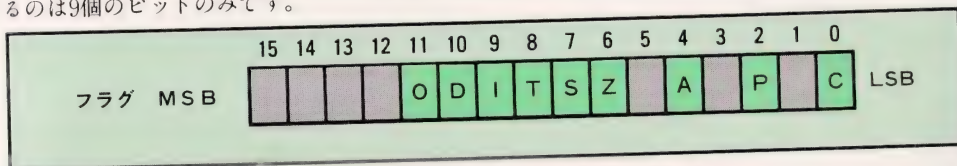
2.7 フラグの役割

8086が14個のレジスタを内蔵していることは前述しました。これらのレジスタにはそれぞれ使用目的が決まっており、各レジスタの名称も目的にそったものがつけられています。

本節では、フラグの用途を説明します。

フラグも16ビットのレジスタですが、16ビットの中で通常使用されるのは9個のビットのみです。

フラグ



上図は、フラグの使用状況を示したものです。図中で空いている所(上位4個のビットおよび第5ビット、第3ビット、第1ビット)は使用されないビットです。

また、図中に「MSB」および「LSB」と書いてありますが、それぞれ、最上位ビットと最下位ビットを表わします。この図では各ビットの上側に15~0の通し番号が書いてあるため左側が最上位ビットであることが明白ですが、この通し番号を省略することも多いため、MSBおよびLSBを明記して、左右どちら側が最上位(最下位)ビットであるのかを明白にしておくのです。

MSB(最上位ビット)

LSB(最下位ビット)

さてフラグの役目ですが、他のレジスタ(たとえばレジスタAX)などのように、単に16ビットのデータを格納しておくために使用することはできません。なぜならば、フラグ各ビットの内容(0または1)は命令の実行結果によってCPUが勝手に書き換えてしまうからです。

次表に各ビットの役目を示します。現時点では、ほとんど理解できないかも知れませんが後で詳しく説明して行きますので御心配なく!

- OF (オーバーフロー・フラグ) — 結果が符号も含めて演算範囲を超えたことを示します。
- DF (ディレクション・フラグ) — スtring命令における操作の方向を指定します。
- IF (インタラプト・イネーブル・フラグ) — 外部割り込みが可能が否かを指定します。
- TF (トラップ・フラグ) — デバッグのためのシングル・ステップ動作を指定します。
- SF (サイン・フラグ) — 演算結果が負(マイナス)になったことを示します。
- ZF (ゼロ・フラグ) — 演算結果がゼロ(0)になったことを示します。
- AF (ハーフ・キャリ・フラグ) — 下位4ビットからキャリが出たことを示します。
- PF (パリティ・フラグ) — 演算結果の「1」のビット数が偶数であることを示します。
- CF (キャリ・フラグ) — MSBからキャリ(桁上げ)が出たことを示します。

3章 簡単なプログラム

3.0 メモリの内容をコピーするプログラム

本章からは、いよいよプログラムをつくって行きます。はじめのうちは、短くて簡単な構造のプログラムから手掛けますので、実際にキーボードをたたいて実行し、動作を確認するようにしてください。

最初のプログラムは、メモリの内容をコピーするプログラムです。とりあえず、プログラム・リストとフローチャートを示します。

```
0000 2E
0001 A00010
0004 2E
0005 A20110
0008 F4
```

```
CS:
MOV AL,[1000]
CS:
MOV [1001],AL
HLT
```

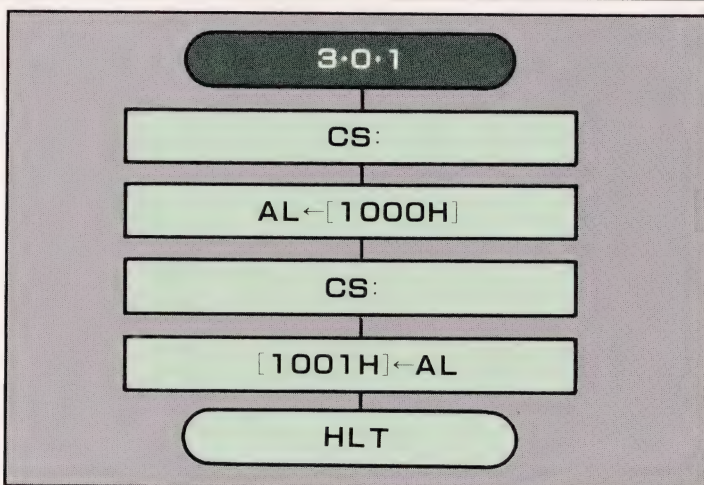
← カレント・コード・セグメントの1000H番地の内容を得る

← カレント・コード・セグメントの1001H番地へ送る

← ホールト

プログラム・リスト↑

フローチャート⇒



リストの内容を検討したり、フローチャートの流れを追ったりするのは後まわしにして、まず、プログラムを入力して走らせてみましょう。

皆さんのPC-9801は機械語モニタのモードになっていますか。まだの方は下図のように入力して、モニタを走らせてください。

```
clear ,&h1800 ← この通り入力して ☒ キーを押す
Ok
def seg=&h1800 ← ☒ キーを押す
Ok
mon ← ☒ キーを押す
h]
↑
カーソル
```

アセンブリ言語のプログラムを入力する場合には、機械語モニタの A (アセンブル) コマンドを利用します。

プログラムは 0000H 番地から入力して行くので、まず次のように入力してください。

A **0** **0** **0** **0** **0** 

ここで 0000H 番地と言うのは、18000H 番地から 64K バイトのセグメントにおける、オフセット・アドレスであることに注意してください。すなわち、実際にプログラムを格納する先頭アドレス (物理アドレス) は、あくまでも 18000H 番地です。

オフセット・アドレス
物理アドレス

8086 について語る場合、このようなオフセット・アドレスと物理アドレスの違いを明確にしておく必要がありますが、本書では前者を 4 桁の 16 進数で、後者を 5 桁の 16 進数で表現することによって両者を区別しています。

さて、A コマンドを入力しますと、画面にアドレスが表示されて入力待ちになります。

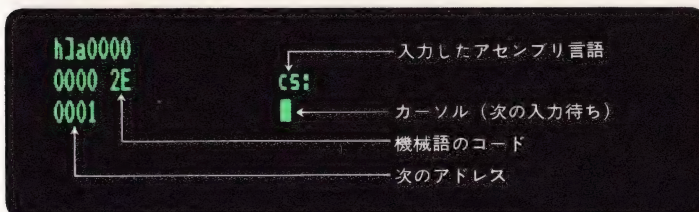


ここで、アセンブリ言語の 1 行目をキーボードから入力します。



間違ったキーを押した場合には  キーでカーソルをもどし、修正することができます。

1 行分のアセンブリ言語を入力し終えた所で  キーを押すと瞬時にアセンブルが行なわれ、いま入力したアセンブリ言語の左側に機械語のコードが表示されます。



プログラムの 2 行目以降も、1 行目と同じ要領で入力して行ってください。

もし、正常でないアセンブリ言語を入力してしまった場合には、 キーを押した時点でブザーが鳴り疑問符が表示されますので、間違っ

た行を入力し直してください。

h]a0000

0000 2E

0001 A00010

0004 2E

0005 A20110

0008 F4

0009

CS:

mov al,[1000]

CS:

mov [1001],al

hlt

█

白い文字の部分が皆さんが入力する
ところ。いずれも最後に  キー
を押してください。

← ここでは  キーのみを押す。

最後の命令を入力し終えると 0009H 番地のアドレスが表示されますが、ここでは  キーのみを押して A (アセンブル) コマンドから抜け出します。

L コマンド

画面上ではうまく入力したように見えても、メモリには正常なコードが入っているかどうかわかりませんので、L (逆アセンブル) コマンドを利用して確認しておきます。

L 0 0 0 0 0 0 0 0 8 

と入力することによって、今入力したプログラムを逆アセンブルして表示します。

h]10000,0008

0000 2E

0001 A00010

0004 2E

0005 A20110

0008 F4

h]█

CS:

MOV

CS:

MOV

HLT

0000H 番地から 0008H 番地までを
逆アセンブル

AL,[1000]

[1001],AL

もし、逆アセンブルの結果が正常でない場合には本項 (3・0) の頭からもう一度やり直してみてください。特に、BASIC から行なう機械語領域の指定 (CLEAR) やセグメントの指定 (DEF SEG) が間違っていると、予想外のアドレスにプログラムを書き込んでいる場合がありますので、よく注意してください。

さて、上図の逆アセンブル結果を見ると、本項 (3・0) の一番最初に提示したプログラム・リストとそっくりなことに気がつきます。これは、本書中に掲載した全てのプログラムのリストが、機械語モニタの L コマンドによって出力されたことを意味します。

P コマンド

ちなみに、この機械語モニタでは、P (プリンタ・スイッチ) コマンドを入力すると、コマンド待ち時のプロンプトが次のように変わり、以後の逆アセンブルはプリンタに出力されます。

h]p

h)█

プログラムの入力終了した時点で、さっそく実行してみたいところですが、その前にデータをセットしておかなくてはなりません。

このプログラムは、カレント・コード・セグメントのオフセット・アドレス1000H番地の内容を1001H番地にコピーするプログラムですから、あらかじめ1000H番地にデータをセットしておきます。データは何でも良いのですが、ここでは55Hを入れておきましょう。

データのセットにはモニタのS（セット）コマンドを利用しますが、

S 1 0 0 0

を入力すると、アドレスとメモリの内容の後にマイナス記号が表示されますので、

5 5

と入力することによって、1000H番地に55Hがセットできます。

```
h]s1000
1000 00-55 ← 1000H番地に55Hをセット
h]■
```

S コマンド

これで、ようやくプログラムを実行することができます。

実行にはモニタのG（ゴー）コマンドを使い、0000H番地からのプログラムを実行するには普通、

G 0 0 0 0

と入力しますが、ここでは少し問題がありそうです。

なぜならば、このプログラムの最後の命令がCPUを停止するためのHLTであるため、0000H番地から逐次実行して行った場合には、0008H番地でCPUが停止（ホールト状態と言う）してしまうからです。ホールト状態から抜け出すためにはリセット・スイッチを押さなければなりません。

[STOP] キーを押しながらリセット・スイッチを押せば、プログラムが破壊されることはありませんが、実行のたびにリセットするのは面倒ですし、精神衛生上もあまり良くありません。そこで、

G 0 0 0 0 0 0 0 8

によって実行することになります。これは、0000H番地から0007H番地までを実行し0008H番地で機械語モニタにもどることを意味します。この0008H番地をブレーク・ポイントと言います。

```
h]g0000,0008
*1800:0008 ← オフセット・アドレス
h]■ ↑      ← セグメントのベース・アドレス
```

G コマンド

ホールト状態

ブレーク・ポイント

プログラムの実行が終了してマシン語モニタにもどる時に表示されるのはブレーク・ポイントのアドレスで、初めの4桁（ここでは1800）がセグメントのベース・アドレス、次の4桁（ここでは0008）がオフセット・アドレスです。

プログラムが正常に動作したか否かを確認しましょう。1000H番地の内容を1001H番地にコピーするプログラムですから1001番地の内容を調べれば正常に動作したかがわかります。

今回は、1000H番地の内容と1001H番地の内容を表示するためにD（ダンプ）コマンドを用いて、

D 1000 1001 

と、入力してみました。

D コマンド

●実行前

```
hjd1000,1001
1000 55 00
hjd
```

U

●実行後

```
hjd1000,1001
1000 55 55
hjd
```

UU

↑ 1001H番地に55Hがコピーされている

実行前と実行後の内容を比較すると、1000H番地の内容が正常に1001H番地へコピーされたのがわかります。

皆さんも正常な実行結果が得られましたか？

もし、うまく結果が得られなかった方、不幸にして暴走させてしまった方もあきらめないで、もう一度チャレンジしてみてください。気力こそが、プログラミング上達の秘訣なのですから！

ここで、このプログラムで使用したアセンブリ言語の命令を解説しておきます。

CS :

●CS :

8086では、データ領域としてレジスタDSでベース・アドレスを指定するカレント・データ・セグメントにデータを置いておくのが普通です。このため、次に説明するMOVなどメモリをアクセスする必要のある命令では、ほとんどの場合にカレント・データ・セグメント中のメモリをアクセスしてしまいます。

ところが、今回のプログラムではカレント・コード・セグメント中のメモリの内容をコピーしなかったため、メモリをアクセスするMOV

の前には全て、CS：を入れておきました。

このように、命令がアクセスすべき本来のセグメント以外のセグメントを指定することを、セグメント・オーバーライドと言います。CS：の他にも、DS：、SS：、ES：の計4種を使用することができます。それぞれの命令と、次に置かれた命令がアクセスするセグメントの関係は次のとおりです。

CS：——カレント・コード・セグメント

DS：——カレント・データ・セグメント

SS：——カレント・スタック・セグメント

ES：——カレント・エクストラ・セグメント

CS：

DS：

SS：

ES：

MOV

MOV

MOVは、レジスタ間またはレジスタとメモリの間でデータをコピーするための命令で、語源はもちろん英語のMOVEです。

今回は、8ビットのレジスタALとメモリの間でデータをコピーするために、次の2つを使ってみました。

```
MOV     AL, [1000H]
```

```
MOV     [1001H], AL
```

前者は1000H番地の内容をレジスタALにコピーするための、後者はレジスタALの内容を1001H番地にコピーするための命令ですが、ここで注意しなくてはいけないのが、本来MOVで扱うメモリがカレント・データ・セグメントであることです。今回はそれぞれの命令の直前にCS：を置いて強制的にカレント・コード・セグメントを指定しました。また、

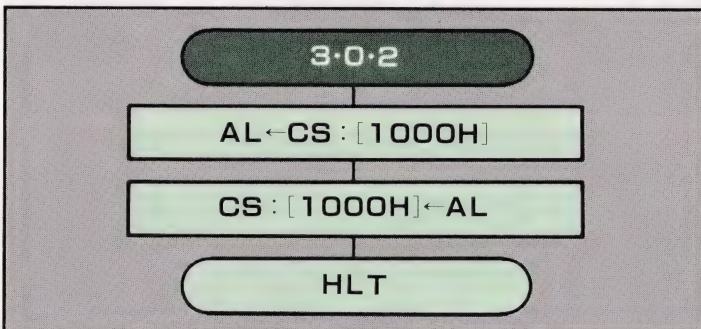
```
CS：
```

```
MOV     AL, [1000H]
```

などと2行に渡るのが不愉快な場合には、アドレス指定の直前にCS：を組み込んで次のように1行にまとめることもできます。

```
MOV     AL, CS:[1000H]
```

この方法で先のプログラム全体を書き換えたものを示します。



←フローチャート


```
0000 2EA00010
0004 2EA20110
0008 F4
```

```
MOV AL,CS:[1000]
MOV CS:[1001],AL
HLT
```

← カレント・コード・セグメントの1000H番地の内容も送る
← カレント・コード・セグメントの1001H番地へ送る
← ホールト

↑プログラム・リスト

しかし、Aコマンドでキーボードから打ち込む時には良いのですが、Lコマンドによって得たプログラム・リストは、全てのセグメント・オーバーライドが2行に分割された形になっています。どちらの表現方法を選ぶのかは趣味の問題もあって一概に決めることはできませんが、私は1行にまとめた方が分かりやすいと思います。

また、本書に掲載したプログラム・リストや命令の中には、16進数の末尾にHがついたものについていないものがありますが、これは機械語モニタ内蔵のアセンブラ/逆アセンブラ(Aコマンド/Lコマンド)がHをつけないで16進数を表現する(16進数以外は扱えない)ためです。したがって、プリンタに出力したもの、およびCRT画面の写真は16進数であっても末尾にHを伴っていません。

16進数の末尾にはHをつけるのが本来の形ですから、本文中の説明(写植や活字によるもの)にはできる限りHを付加して区別するようにしました。

HLT

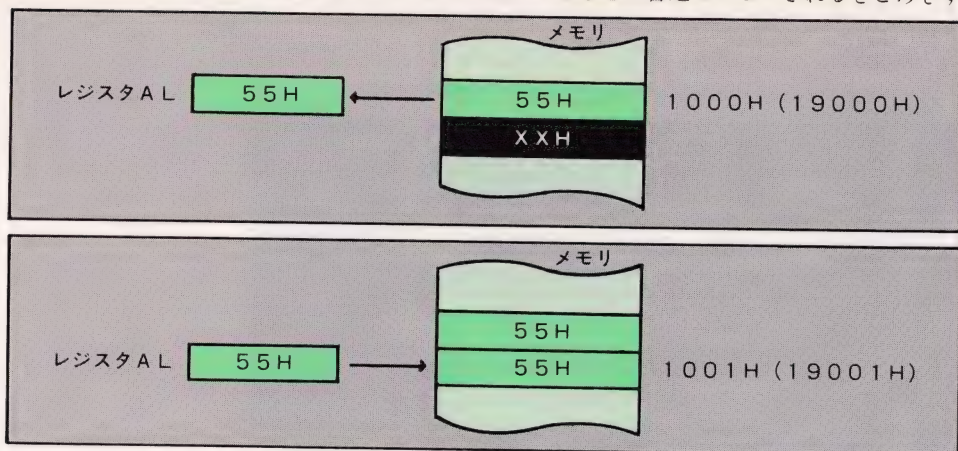
ホールト状態

●HLT

8086は、この命令を実行するとホールト状態に入ります。ホールト状態ではキーボードからの入力なども全て受け付けられないため、リセット・スイッチを押さなくてはなりません。

HLTの語源は、HALT(停止の意)です。

次の図は、本プログラムを実行することによって、1000H番地に格納されている55Hが1001H番地にコピーされるところです。



最後に、本節のプログラムを入力し実行する手順を簡単にまとめておきます。

```
clear ,&h1800
```

```
Ok
```

```
def seg=&h1800
```

```
Ok
```

```
mon
```

```
h]a0000
```

```
0000 2E
```

```
0001 A00010
```

```
0004 2E
```

```
0005 A20110
```

```
0008 F4
```

```
0009
```

```
h]l0000,0008
```

```
0000 2E
```

```
0001 A00010
```

```
0004 2E
```

```
0005 A20110
```

```
0008 F4
```

```
h]s1000
```

```
1000 00-55
```

```
h]d1000,1001
```

```
1000 55 00
```

```
h]g0000,0008
```

```
*1800:0008
```

```
h]d1000,1001
```

```
1000 55 55
```

```
h]■
```

機械語モニタを起動

```
CS:
```

```
mov al,[1000]
```

```
CS:
```

```
mov [1001],al
```

```
hit
```

アセンブリ言語のプログラムを入力

```
CS:
```

```
MOV AL,[1000]
```

```
CS:
```

```
MOV [1001],AL
```

```
HLT
```

逆アセンブルで確認

データをセット

U } データを確認

プログラムを実行

UU } 実行結果を確認

白ヌキ文字が皆さんが入力するところです。

3.1 PC-9801の画面構成

PC-9801ではCRT画面上にキャラクタ（文字）を表示するために、VRAM（Video-RAM）方式を採用しています。もちろん、個々のキャラクタにカラーをつけたり、640×400ドットのフル・グラフィック機能を利用することなどもできますが、ここでは話を簡単にするため、単にキャラクタを表示することのみを考えてみます。

VRAM方式

VRAM方式では、CRTに表示する1画面分に対応した内容をメモリ上にも確保しておかなければなりません。この、メモリ上に確保した領域をVRAM（Video-RAM）と言い、CRT画面の表示は全てVRAMを通して行ないます。すなわち、VRAMに適当なキャラクタ・コードを書き込めば、即、そのキャラクタがCRT画面上の対応する位置に表示されることになります。

それでは、実際にVRAMを操作して、CRT画面にキャラクタを表示してみましょう。

PC-9801のVRAMは、A0000H番地からの約4Kバイトに位置しており、80桁モードの場合には2番地おきに、40桁モードの場合には4番地おきに使用します。奇数アドレスは漢字を表示するために使用するのですが、ここでは空けておきます。

ためしに、A0000H番地にXのキャラクタ・コードである58Hを書き込んでみましょう。

Cコマンド

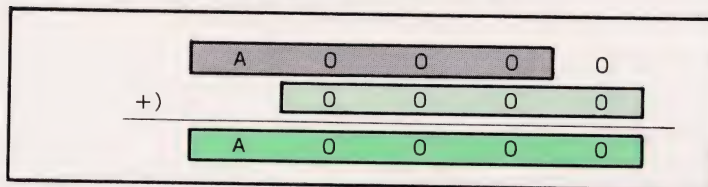
まず、モニタのC（チェンジ・セグメント）コマンドによって、モニタで扱うセグメントをA0000H～AFFFFFH番地に変更します。次のように入力してください。

C A 0 0 0 0

hJca000
hJ■

セグメントをA0000H～AFFFFFH番地に変更

この状態で、0000H番地に58Hを書き込めば、実際にはA0000H番地に58Hを書き込んだことになります。



書き込みにはモニタのSコマンド等を使用します。

h]ca000

h]s0000

0000 68-58

h]■

← A 0 0 0 0 H 番地からのデータ書き込み命令

← 書き込むべきデータ

← 書き込み前の A 0 0 0 0 H 番地の内容

どうですか？ CRT画面のホーム・ポジション（LOCATE 0,0の位置）にXの文字が現われたでしょう。

同じようにして、A 0 0 0 2 H 番地、A 0 0 0 4 H 番地、A 0 0 0 6 H 番地、A 0 0 0 8 H 番地……にも58Hを書き込んでみてください。

XXXXXXXX

ホーム・ポジションからXが表示される

h]s0000

0000 68-58

h]s0002

0002 50-58

h]s0004

0004 63-58

h]s0006

0006 61-58

h]s0008

0008 30-58

h]s000a

000A 30-58

h]s000c

000C 30-58

h]s000e

000E 20-58

h]■

← VRAMに2番地おきに58Hを書き込んで行く

load ' save goto list run save ' key print edit . cont

次ページのBASICによるプログラムおよび実行サンプルは、CRT画面全体をランダムなキャラクタで埋めるものです。もちろん、画面に文字を表示する命令（PRINT および WRITE など）は一切使用せずに、いま説明したVRAMに直接キャラクタ・コードを書き込んでいきます（POKE）。

POKE

リストを見てもわかるように、画面上の座標（X、Y）に対応する80桁モードでのオフセット・アドレス（OFFSET）は、

$OFFSET = Y * 160 + X * 2$

の式によって求めることができますが、これは、扱うセグメントのベース・アドレスがA 0 0 0 0 H 番地の場合にのみ有効です。

次のページから2ページに渡って、CRT画面とPC-9801の標準的なテキストVRAMの対応表を掲載します。左ページは80桁モードの場合、右のページは40桁モードの場合です。

なお、これらの対応表は、下記のプログラムによって、PC-8822（ドット・マトリクス漢字プリンタ）より打ち出したものです。

```

100 LPRINT CHR$( &H1B ); "Q"; CHR$( &H1B ); "!"; CHR$( 12 );
110 LPRINT TAB( 51 ); "TEXT VRAM FOR 80 CHARACTERS MODE"
120 LPRINT
130 LPRINT "      0      1      2      3      4      5      6      7      8      9";
140 LPRINT "          71    72    73    74    75    76    77    78    79";
150 LPRINT "      "; STRING$( 122, "-" )
160 FOR Y=0 TO 24
170   LPRINT USING "## "; Y;
180   FOR X=0 TO 9
190     LPRINT "A"; RIGHT$( "000" + HEX$( Y*160 + X*2 ), 4 ); " ";
200   NEXT X
210   LPRINT "..... ";
220   FOR X=71 TO 79
230     LPRINT "A"; RIGHT$( "000" + HEX$( Y*160 + X*2 ), 4 ); " ";
240   NEXT X
250   LPRINT
260 NEXT Y
270 LPRINT "      "; STRING$( 122, "-" )
280 '
290 LPRINT CHR$( &H1B ); "Q"; CHR$( &H1B ); "!"; CHR$( 12 );
300 LPRINT TAB( 51 ); "TEXT VRAM FOR 40 CHARACTERS MODE"
310 LPRINT
320 LPRINT "      0      1      2      3      4      5      6      7      8      9";
330 LPRINT "          31    32    33    34    35    36    37    38    39";
340 LPRINT "      "; STRING$( 122, "-" )
350 FOR Y=0 TO 24
360   LPRINT USING "## "; Y;
370   FOR X=0 TO 9
380     LPRINT "A"; RIGHT$( "000" + HEX$( Y*160 + X*4 ), 4 ); " ";
390   NEXT X
400   LPRINT "..... ";
410   FOR X=31 TO 39
420     LPRINT "A"; RIGHT$( "000" + HEX$( Y*160 + X*4 ), 4 ); " ";
430   NEXT X
440   LPRINT
450 NEXT Y
460 LPRINT "      "; STRING$( 122, "-" )
470 END

```

見出しのプリント

縦座標の表示

横座標の表示

オフセット・アドレスを4桁の16進数に

TEXT VRAM FOR 80 CHARACTERS MODE

	0	1	2	3	4	5	6	7	8	9	71	72	73	74	75	76	77	78	79	
0	A0000	A0002	A0004	A0006	A0008	A000A	A000C	A000E	A0010	A0012		A008E	A0090	A0092	A0094	A0096	A0098	A009A	A009C	A009E
1	A00A0	A00A2	A00A4	A00A6	A00A8	A00AA	A00AC	A00AE	A00B0	A00B2		A012E	A0130	A0132	A0134	A0136	A0138	A013A	A013C	A013E
2	A0140	A0142	A0144	A0146	A0148	A014A	A014C	A014E	A0150	A0152		A01CE	A01D0	A01D2	A01D4	A01D6	A01D8	A01DA	A01DC	A01DE
3	A01E0	A01E2	A01E4	A01E6	A01E8	A01EA	A01EC	A01EE	A01F0	A01F2		A026E	A0270	A0272	A0274	A0276	A0278	A027A	A027C	A027E
4	A0280	A0282	A0284	A0286	A0288	A028A	A028C	A028E	A0290	A0292		A030E	A0310	A0312	A0314	A0316	A0318	A031A	A031C	A031E
5	A0320	A0322	A0324	A0326	A0328	A032A	A032C	A032E	A0330	A0332		A03AE	A03B0	A03B2	A03B4	A03B6	A03B8	A03BA	A03BC	A03BE
6	A03C0	A03C2	A03C4	A03C6	A03C8	A03CA	A03CC	A03CE	A03D0	A03D2		A044E	A0450	A0452	A0454	A0456	A0458	A045A	A045C	A045E
7	A0460	A0462	A0464	A0466	A0468	A046A	A046C	A046E	A0470	A0472		A04EE	A04F0	A04F2	A04F4	A04F6	A04F8	A04FA	A04FC	A04FE
8	A0500	A0502	A0504	A0506	A0508	A050A	A050C	A050E	A0510	A0512		A058E	A0590	A0592	A0594	A0596	A0598	A059A	A059C	A059E
9	A05A0	A05A2	A05A4	A05A6	A05A8	A05AA	A05AC	A05AE	A05B0	A05B2		A062E	A0630	A0632	A0634	A0636	A0638	A063A	A063C	A063E
10	A0640	A0642	A0644	A0646	A0648	A064A	A064C	A064E	A0650	A0652		A06CE	A06D0	A06D2	A06D4	A06D6	A06D8	A06DA	A06DC	A06DE
11	A06E0	A06E2	A06E4	A06E6	A06E8	A06EA	A06EC	A06EE	A06F0	A06F2		A076E	A0770	A0772	A0774	A0776	A0778	A077A	A077C	A077E
12	A0780	A0782	A0784	A0786	A0788	A078A	A078C	A078E	A0790	A0792		A080E	A0810	A0812	A0814	A0816	A0818	A081A	A081C	A081E
13	A0820	A0822	A0824	A0826	A0828	A082A	A082C	A082E	A0830	A0832		A08AE	A08B0	A08B2	A08B4	A08B6	A08B8	A08BA	A08BC	A08BE
14	A08C0	A08C2	A08C4	A08C6	A08C8	A08CA	A08CC	A08CE	A08D0	A08D2		A094E	A0950	A0952	A0954	A0956	A0958	A095A	A095C	A095E
15	A0960	A0962	A0964	A0966	A0968	A096A	A096C	A096E	A0970	A0972		A09EE	A09F0	A09F2	A09F4	A09F6	A09F8	A09FA	A09FC	A09FE
16	A0A00	A0A02	A0A04	A0A06	A0A08	A0A0A	A0A0C	A0A0E	A0A10	A0A12		A0A8E	A0A90	A0A92	A0A94	A0A96	A0A98	A0A9A	A0A9C	A0A9E
17	A0AA0	A0AA2	A0AA4	A0AA6	A0AA8	A0AAA	A0AAC	A0AAE	A0AB0	A0AB2		A0B2E	A0B30	A0B32	A0B34	A0B36	A0B38	A0B3A	A0B3C	A0B3E
18	A0B40	A0B42	A0B44	A0B46	A0B48	A0B4A	A0B4C	A0B4E	A0B50	A0B52		A0BCE	A0BD0	A0BD2	A0BD4	A0BD6	A0BD8	A0BDA	A0BDC	A0BDE
19	A0BE0	A0BE2	A0BE4	A0BE6	A0BE8	A0BEA	A0BEC	A0BEE	A0BF0	A0BF2		A0C6E	A0C70	A0C72	A0C74	A0C76	A0C78	A0C7A	A0C7C	A0C7E
20	A0C80	A0C82	A0C84	A0C86	A0C88	A0C8A	A0C8C	A0C8E	A0C90	A0C92		A0D0E	A0D10	A0D12	A0D14	A0D16	A0D18	A0D1A	A0D1C	A0D1E
21	A0D20	A0D22	A0D24	A0D26	A0D28	A0D2A	A0D2C	A0D2E	A0D30	A0D32		A0DAE	A0DB0	A0DB2	A0DB4	A0DB6	A0DB8	A0DBA	A0DBC	A0DBE
22	A0DC0	A0DC2	A0DC4	A0DC6	A0DC8	A0DCA	A0DCC	A0DCE	A0DD0	A0DD2		A0E4E	A0E50	A0E52	A0E54	A0E56	A0E58	A0E5A	A0E5C	A0E5E
23	A0E60	A0E62	A0E64	A0E66	A0E68	A0E6A	A0E6C	A0E6E	A0E70	A0E72		A0EEE	A0EF0	A0EF2	A0EF4	A0EF6	A0EF8	A0EFA	A0EFC	A0EFE
24	A0F00	A0F02	A0F04	A0F06	A0F08	A0F0A	A0F0C	A0F0E	A0F10	A0F12		A0F8E	A0F90	A0F92	A0F94	A0F96	A0F98	A0F9A	A0F9C	A0F9E

TEXT VRAM FOR 40 CHARACTERS MODE

0	1	2	3	4	5	6	7	8	9	31	32	33	34	35	36	37	38	39		
0	A0000	A0004	A0008	A000C	A0010	A0014	A0018	A001C	A0020	A0024		A007C	A0080	A0084	A0088	A008C	A0090	A0094	A0098	A009C
1	A00A0	A00A4	A00A8	A00AC	A00B0	A00B4	A00B8	A00BC	A00C0	A00C4		A011C	A0120	A0124	A0128	A012C	A0130	A0134	A0138	A013C
2	A0140	A0144	A0148	A014C	A0150	A0154	A0158	A015C	A0160	A0164		A01BC	A01C0	A01C4	A01C8	A01CC	A01D0	A01D4	A01D8	A01DC
3	A01E0	A01E4	A01E8	A01EC	A01F0	A01F4	A01F8	A01FC	A0200	A0204		A025C	A0260	A0264	A0268	A026C	A0270	A0274	A0278	A027C
4	A0280	A0284	A0288	A028C	A0290	A0294	A0298	A029C	A02A0	A02A4		A02FC	A0300	A0304	A0308	A030C	A0310	A0314	A0318	A031C
5	A0320	A0324	A0328	A032C	A0330	A0334	A0338	A033C	A0340	A0344		A039C	A03A0	A03A4	A03A8	A03AC	A03B0	A03B4	A03B8	A03BC
6	A03C0	A03C4	A03C8	A03CC	A03D0	A03D4	A03D8	A03DC	A03E0	A03E4		A043C	A0440	A0444	A0448	A044C	A0450	A0454	A0458	A045C
7	A0460	A0464	A0468	A046C	A0470	A0474	A0478	A047C	A0480	A0484		A04DC	A04E0	A04E4	A04E8	A04EC	A04F0	A04F4	A04F8	A04FC
8	A0500	A0504	A0508	A050C	A0510	A0514	A0518	A051C	A0520	A0524		A057C	A0580	A0584	A0588	A058C	A0590	A0594	A0598	A059C
9	A05A0	A05A4	A05A8	A05AC	A05B0	A05B4	A05B8	A05BC	A05C0	A05C4		A061C	A0620	A0624	A0628	A062C	A0630	A0634	A0638	A063C
10	A0640	A0644	A0648	A064C	A0650	A0654	A0658	A065C	A0660	A0664		A06BC	A06C0	A06C4	A06C8	A06CC	A06D0	A06D4	A06D8	A06DC
11	A06E0	A06E4	A06E8	A06EC	A06F0	A06F4	A06F8	A06FC	A0700	A0704		A075C	A0760	A0764	A0768	A076C	A0770	A0774	A0778	A077C
12	A0780	A0784	A0788	A078C	A0790	A0794	A0798	A079C	A07A0	A07A4		A07FC	A0800	A0804	A0808	A080C	A0810	A0814	A0818	A081C
13	A0820	A0824	A0828	A082C	A0830	A0834	A0838	A083C	A0840	A0844		A089C	A08A0	A08A4	A08A8	A08AC	A08B0	A08B4	A08B8	A08BC
14	A08C0	A08C4	A08C8	A08CC	A08D0	A08D4	A08D8	A08DC	A08E0	A08E4		A093C	A0940	A0944	A0948	A094C	A0950	A0954	A0958	A095C
15	A0960	A0964	A0968	A096C	A0970	A0974	A0978	A097C	A0980	A0984		A09DC	A09E0	A09E4	A09E8	A09EC	A09F0	A09F4	A09F8	A09FC
16	A0A00	A0A04	A0A08	A0A0C	A0A10	A0A14	A0A18	A0A1C	A0A20	A0A24		A0A7C	A0A80	A0A84	A0A88	A0A8C	A0A90	A0A94	A0A98	A0A9C
17	A0AA0	A0AA4	A0AA8	A0AAC	A0AB0	A0AB4	A0AB8	A0ABC	A0AC0	A0AC4		A0B1C	A0B20	A0B24	A0B28	A0B2C	A0B30	A0B34	A0B38	A0B3C
18	A0B40	A0B44	A0B48	A0B4C	A0B50	A0B54	A0B58	A0B5C	A0B60	A0B64		A0BBC	A0BC0	A0BC4	A0BC8	A0BCC	A0BD0	A0BD4	A0BD8	A0BDC
19	A0BE0	A0BE4	A0BE8	A0BEC	A0BF0	A0BF4	A0BF8	A0BFC	A0C00	A0C04		A0C5C	A0C60	A0C64	A0C68	A0C6C	A0C70	A0C74	A0C78	A0C7C
20	A0C80	A0C84	A0C88	A0C8C	A0C90	A0C94	A0C98	A0C9C	A0CA0	A0CA4		A0CFC	A0D00	A0D04	A0D08	A0D0C	A0D10	A0D14	A0D18	A0D1C
21	A0D20	A0D24	A0D28	A0D2C	A0D30	A0D34	A0D38	A0D3C	A0D40	A0D44		A0D9C	A0DA0	A0DA4	A0DA8	A0DAC	A0DB0	A0DB4	A0DB8	A0DBC
22	A0DC0	A0DC4	A0DC8	A0DCC	A0DD0	A0DD4	A0DD8	A0DDC	A0DE0	A0DE4		A0E3C	A0E40	A0E44	A0E48	A0E4C	A0E50	A0E54	A0E58	A0E5C
23	A0E60	A0E64	A0E68	A0E6C	A0E70	A0E74	A0E78	A0E7C	A0E80	A0E84		A0EDC	A0EE0	A0EE4	A0EE8	A0EEC	A0EF0	A0EF4	A0EF8	A0EFC
24	A0F00	A0F04	A0F08	A0F0C	A0F10	A0F14	A0F18	A0F1C	A0F20	A0F24		A0F7C	A0F80	A0F84	A0F88	A0F8C	A0F90	A0F94	A0F98	A0F9C

3.2 メッセージを表示するプログラム

VRAMの構造がわかったところで、さっそくCRT画面への表示を応用したプログラムをつくってみましょう。同時にデータ・セグメント・レジスタ（DS）の一般的な使用方法も紹介しますので、十分に理解しておいてください。

前節3.1では機械語モニタのSコマンドによってVRAMにキャラクタ・コードを書き込んでみましたが、ここではアセンブリ言語のプログラムを実行することによって、CRT画面のホーム・ポジションから下記のメッセージを表示することを考えましょう。

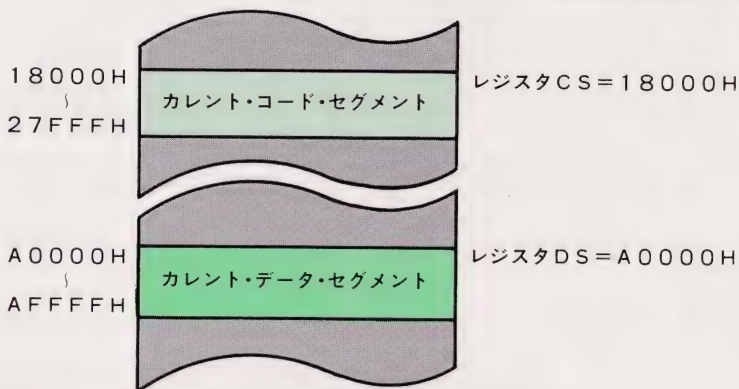
PC-9801

使用する命令はMOVのみですから、プログラムは非常に簡単な構造をしています。VRAMのアドレス（A0000H番地から後）をアクセスするために、データ・セグメント・レジスタ（DS）を設定しています。プログラム冒頭にある、

```
MOV      AX, A000H
MOV      DS, AX
```

の2ステップで、レジスタDSにA000Hをセットしていますが、これ以降のMOVで指定する全てのアドレスは、A0000H番地をベース・アドレスとするセグメント中のオフセット・アドレスとして扱われます。すなわち、MOVで指定するアドレスにA0000Hを加えたものが、実際にアクセスを行なう物理アドレスとなります。

次図は、本プログラムにおけるセグメントの割り当て状況です。



メッセージの表示は、VRAMにキャラクタ・コードを送り込んで行くことで行ないますから、あらかじめ、表示したいキャラクタのキャラクタ・コードを調べておく必要があります（右表）。ある程度きちんとしたアセンブラであれば、

```
MOV     AL, 'P'
```

などで、レジスタALにPのキャラクタ・コードを格納する命令を表現することができますが、内蔵アセンブラ（Aコマンド）では、このような記述は許されていませんから、上記の例であれば、

```
MOV     AL, 50H
```

と表現します。ただし、Aコマンドから打ち込む場合には16進数末尾のHを省略しなければなりません。

アセンブリ言語のプログラム・リストを見て、『キャラクタ・コードを一度レジスタを介してVRAM（メモリ）へ送り込んでいる点を不合理に思った方もおられると思います。実は、

```
MOV     AL, 50H
MOV     [0000H], AL
```

のかわりに、

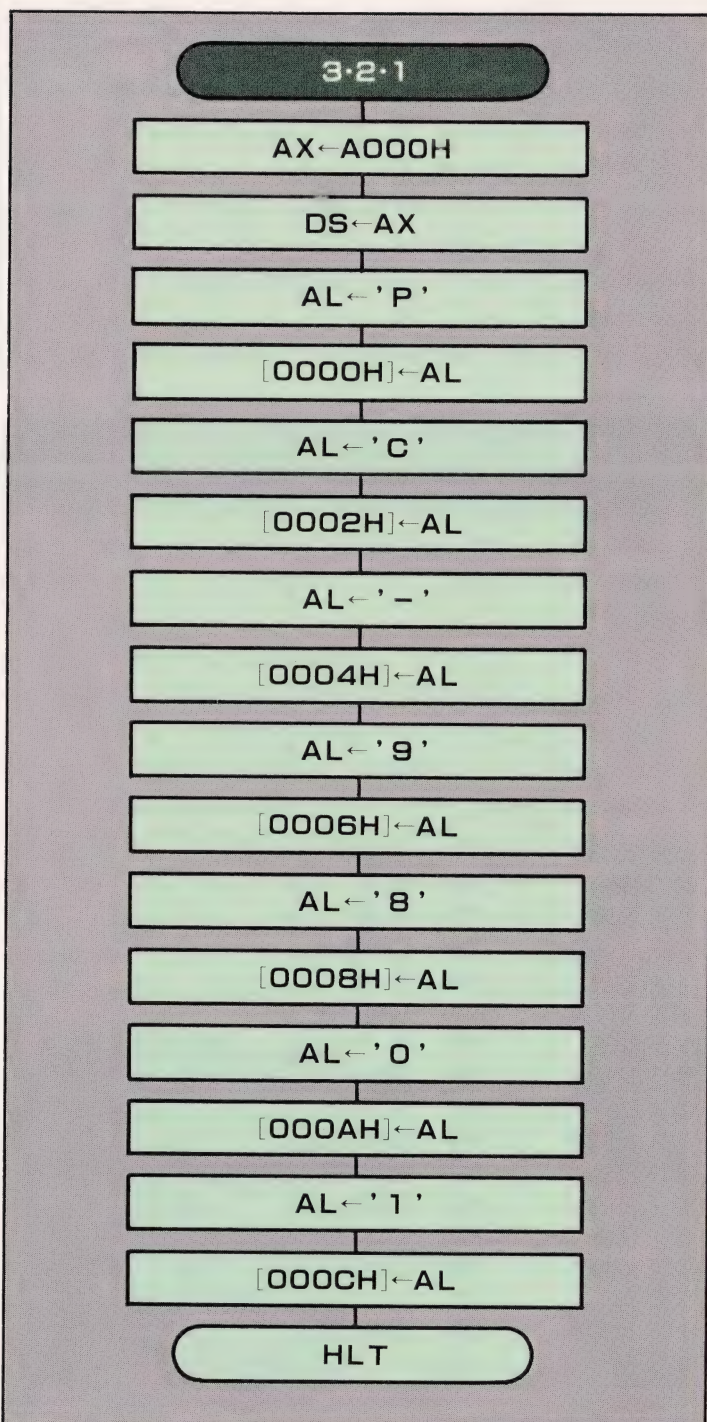
```
MOV     BYTE [0000H], 50H
```

と表現することもできるのですが、機械語にアSEMBルしたときのバイト数も同じですし、わかりやすい前者を採用しました。ちなみに、アドレス指定の左大かっこの前のBYTEは、転送するデータがバイト・データであることを示し、ワード・データの場合はWORDをつけます。

キャラクタ	キャラクタコード
P	50H
C	43H
—	2DH
9	39H
8	38H
0	30H
1	31H

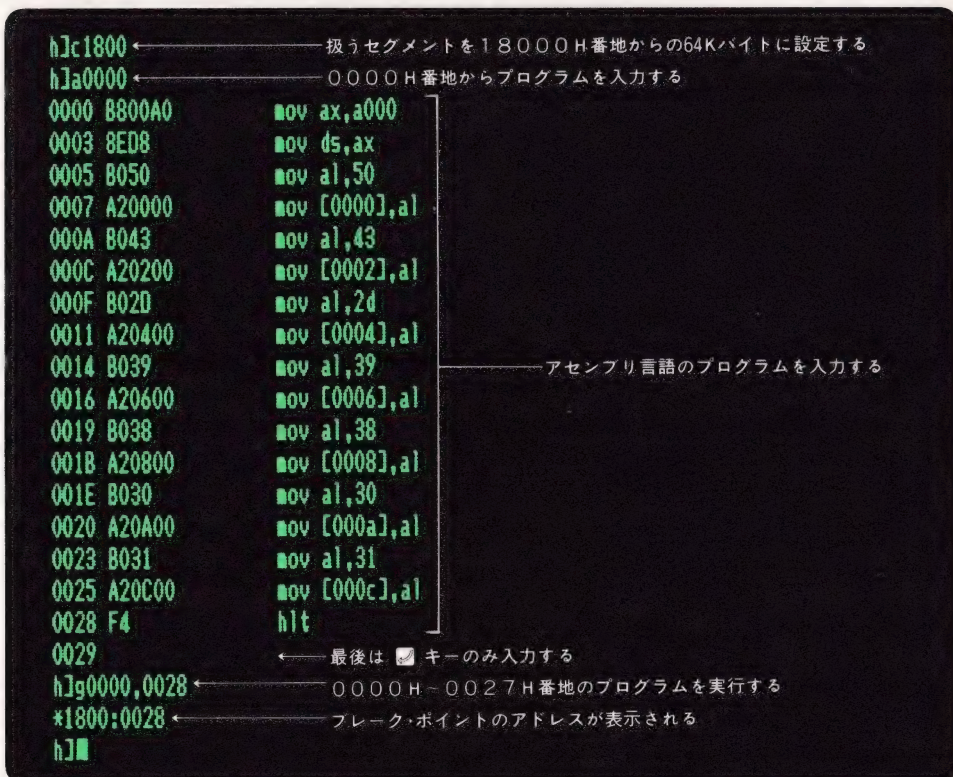
BYTE
WORD

0000 B800A0	MOV	AX,A000	←データ・セグメント・レジスタ(DS)にA000Hをセットする
0003 8ED8	MOV	DS,AX	
0005 B050	MOV	AL,50	
0007 A20000	MOV	[0000],AL	←A00000H番地に50H (P)を送る
000A B043	MOV	AL,43	
000C A20200	MOV	[0002],AL	←A00020H番地に43H (C)を送る
000F B02D	MOV	AL,2D	
0011 A20400	MOV	[0004],AL	←A00040H番地に2DH (—)を送る
0014 B039	MOV	AL,39	
0016 A20600	MOV	[0006],AL	←A00060H番地に39H (9)を送る
0019 B038	MOV	AL,38	
001B A20800	MOV	[0008],AL	←A00080H番地に38H (8)を送る
001E B030	MOV	AL,30	
0020 A20A00	MOV	[000A],AL	←A000A0H番地に30H (0)を送る
0023 B031	MOV	AL,31	
0025 A20C00	MOV	[000C],AL	←A000C0H番地に31H (1)を送る
0028 F4	HLT		←ホールド



フローチャート➡

皆さんも、いくつかのプログラムを通じて、アセンブリ言語によるプログラムの入力方法および実行方法を十分に理解されたことと思いますが、ここでもう一度実行手順を示しておきます。



最初に、機械語モニターから、

☒

を入力して扱うセグメントのベース・アドレスを設定し直していますが、複数のセグメントを扱うプログラムでは、万一の事も考え、ベース・アドレスを確認する意味も込めて、たびたびセグメントを指定し直す習慣をつけておくに便利です。

私も以前、VRAMを扱うプログラムのデバッグ中にA0000H番地からの内容をダンプしようと思い無意識のうちに、

☒

を実行後、必死になってVRAMに機械語のプログラムを書き込んだ苦い経験があります。

プログラムの実行時には、機械語プログラムのアドレスに合わせて、

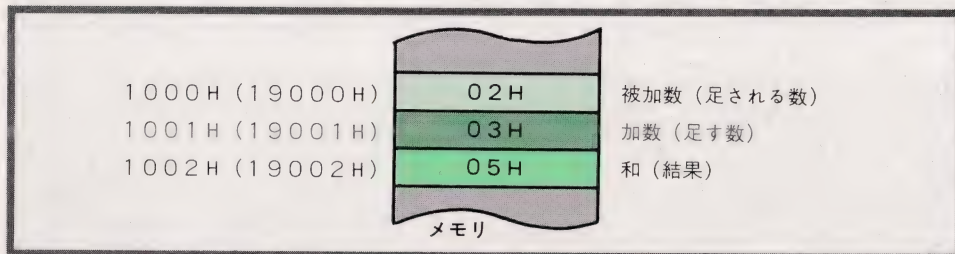
☒

を入力します。

3.3 8ビットの加算を行なうプログラム

アセンブリ言語で加算を行なうプログラムをつくとどうなるでしょうか。8ビットでは255以下のデータしか扱えず少しもの足りないかも知れませんが、まずは8ビットの加算命令を使ってみましょう。

メモリ上の2つの数値の和を求めて、別のメモリに格納することにし、データを入れるために3バイトのメモリを割り当てなければなりません。ここでは次のように決めておきましょう。



このように、カレント・コード・セグメントの1000H番地 (物理アドレスは19000H番地) に置かれた数値と1001H番地に置かれた数値の和を求めて、1002H番地に格納するプログラムをつくることにしました。

被加数および加数は、モニタのSコマンド等によってセットしますが、ここでは一応02Hと03Hを足してみることにします。もちろん皆さんは好きな数値をセットして加算を行なってください。

次にプログラム・リストを示します。

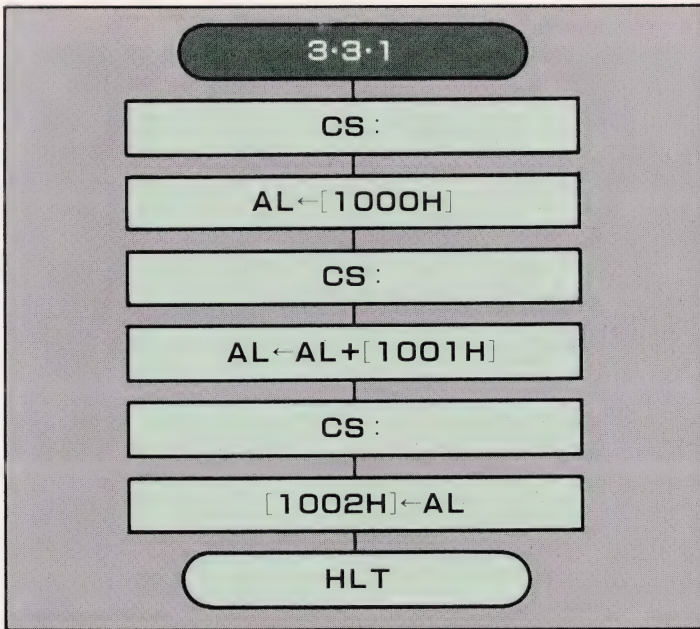
0000 2E	CS:		
0001 A00010	MOV	AL, [1000]	— カレント・コード・セグメントの1000H番地から被加数を得る
0004 2E	CS:		
0005 02060110	ADD	AL, [1001]	— レジスタALの被加数に1001H番地の加数を加える
0009 2E	CS:		
000A A20210	MOV	[1002], AL	— 結果を1002H番地に格納する
000D F4	HLT		— ホールト

ほとんどの命令は、おなじみのMOVですが、0005H番地の

ADD

ADD AL, [1001H]

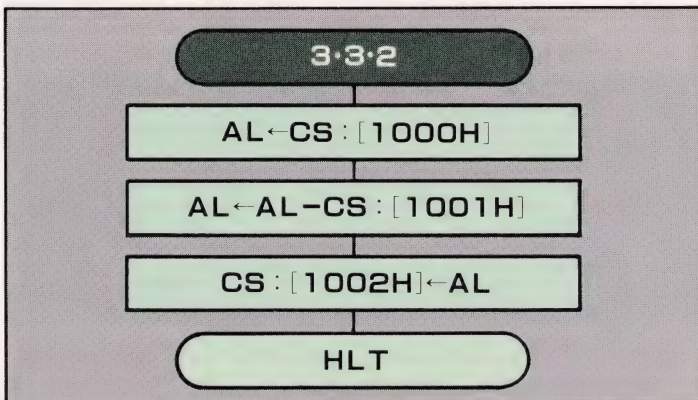
だけは、はじめての命令です。この命令を実行することによって、レジスタALの内容に1001H番地の内容が加えられます。本来1001H番地はカレント・データ・セグメント中のメモリが指定されるはずですが、MOV等と同じように、直前にCS:を置いて、カレント・コード・セグメントを参照させています。



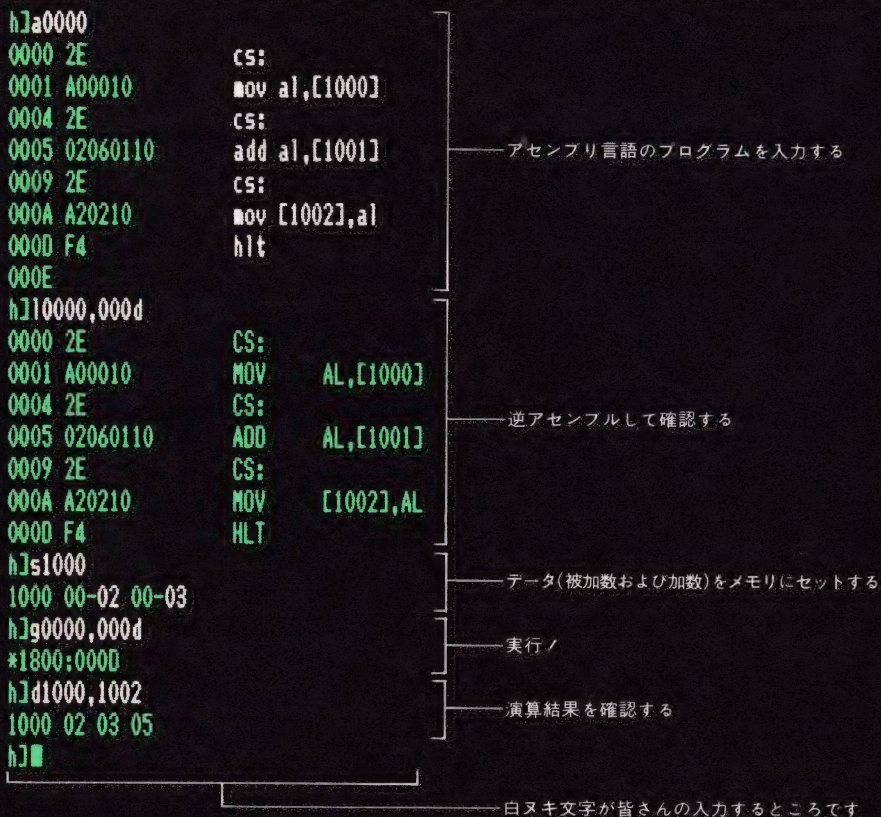
8086には加算命令と同様の使い方ができる減算命令（SUB）も用意されています。この減算命令を用いて、いまの加算プログラムを8ビットの減算を行なうプログラムに変更したものを示します。

SUB

0000 2E	CS:		
0001 A00010	MOV	AL, [1000]	— カレント・セグメントの1000H番地から減算数を得る
0004 2E	CS:		
0005 2A060110	SUB	AL, [1001]	— レジスタALの被減数から1001H番地の減数を減じる
0009 2E	CS:		
000A A20210	MOV	[1002], AL	— 結果を1002H番地に格納する
000D F4	HLT		— ホールト



下図は、プログラムの入力方法および実行方法です。



3.4 16ビットの加減算を行なうプログラム

8ビットの演算ができるようになったところで、今度は16ビットの加減算に挑戦しましょう。

8ビットのときは方法をかえて、2つの数の和と差を一度で求めるプログラムにしてみました。

データ・エリアとしては、

被演算項 (2バイト)

演算項 (2バイト)

和 (2バイト)

差 (2バイト)

の、計8バイト分を確保することが必要なので、下図のように設定しました。

1000H (19000H)	0FH	被演算項の下位バイト
1001H (19001H)	10H	被演算項の上位バイト
1002H (19002H)	00H	演算項の下位バイト
1003H (19003H)	05H	演算項の上位バイト
1004H (19004H)	0FH	和の下位バイト
1005H (19005H)	15H	和の上位バイト
1006H (19006H)	0FH	差の下位バイト
1007H (19007H)	0BH	差の上位バイト

ここでは、2つの数値すなわち100FHと0500Hをデータ・エリアにセットして、和および差を求めることにしました。2つの数値をBASICのダイレクト・モードで10進数に変換してみましょう。

```
print "100fh =";&h100f
```

```
100fh = 4111.
```

```
Ok
```

```
print "0500h =";&h0500
```

```
0500h = 1280.
```

```
Ok
```

100FHは10進数では4111に相当する

0500Hは10進数では1280に相当する

和および差は10進数の場合と同じ要領で計算することができますが、9ではなくFを越えた時点で桁が上(下)がることに注意してください。

加算		減算	
	100FH		100FH
+) 0500H		-) 0500H	
	150FH		0B0FH

一応、上記の筆算の結果が正しいかどうかを確認します。

```
print '100fh + 0500h = ';right$('000'+hex$(&h100f+&h0500),4);'h';
100fh + 0500h = 150Fh
Ok
print '100fh - 0500h = ';right$('000'+hex$(&h100f-&h0500),4);'h';
100fh - 0500h = 0B0Fh
Ok
■
```

以上の仕様によってつくった16ビット加減算のプログラム・リストを示します。

0000 2E	CS:		
0001 A10010	MOV	AX,[1000]	1000H~1001H番地に渡る被加数を得る
0004 2E	CS:		
0005 03060210	ADD	AX,[1002]	レジスタAXの被加数に1002H~1003H番地に渡る加数を加える
0009 2E	CS:		
000A A30410	MOV	[1004],AX	和を1004H~1005H番地に格納する
000D 2E	CS:		
000E A10010	MOV	AX,[1000]	1000H~1001H番地に渡る被減数を得る
0011 2E	CS:		
0012 2B060210	SUB	AX,[1002]	レジスタAXの被減数から1002H~1003H番地に渡る減数を減じる
0016 2E	CS:		
0017 A30610	MOV	[1006],AX	差を1006H~1007H番地に格納する
001A F4	HLT		ホールド

プログラムの流れ自体は8ビットの場合とほとんど同じで、最初に加算命令 (ADD) で和を求めた後、減算命令 (SUB) を用い、和を求めたのと同じ手順で差を求めています。

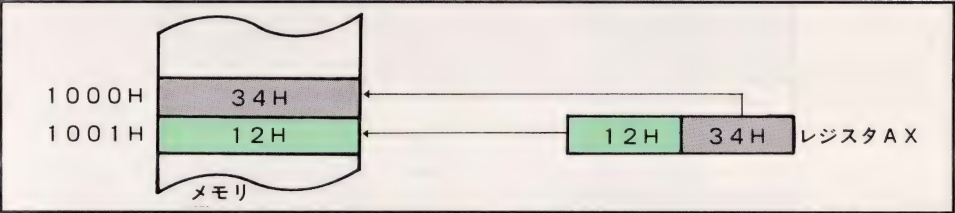
もちろん、扱う全てのデータが16ビット長ですから、MOV、ADD、SUBなどの命令も全て16ビットのための命令ですし、レジスタも16ビット長のレジスタAXを使用しています。先にも述べたように、16ビット長のデータをワード・データなどと呼びますが、8086を含む80系のCPU(8080、8085、Z80など)ではメモリ上でのワード・データの置かれ方(ロケーション)に特色があります。この特色は、逆ワードなどと呼ばれるもので、たとえば、

MOV CS:[1000H],AX

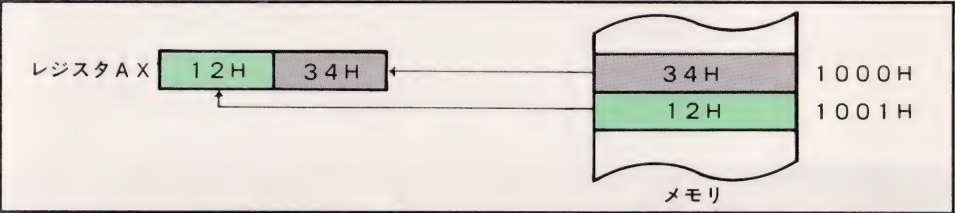
ワード・データ

逆ワード

を実行して、レジスタAXのワード・データをカレント・コード・セグメントの1000H~1001H番地に格納した場合に、レジスタAXの下位バイト（レジスタAL）が1000H番地に、レジスタAXの上位バイト（レジスタAH）が1001H番地に、それぞれコピーされてしまいます。下図はレジスタAXに1234Hが入っていた場合の逆ワードによるコピー例です。



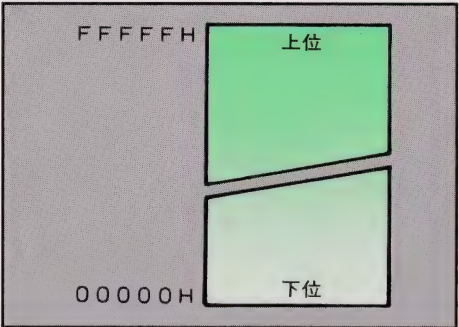
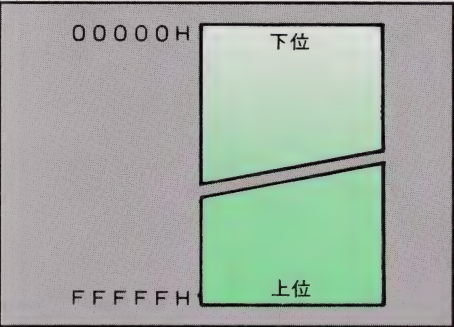
逆にメモリからレジスタへワード・データを送る場合にも、逆ワードになります。



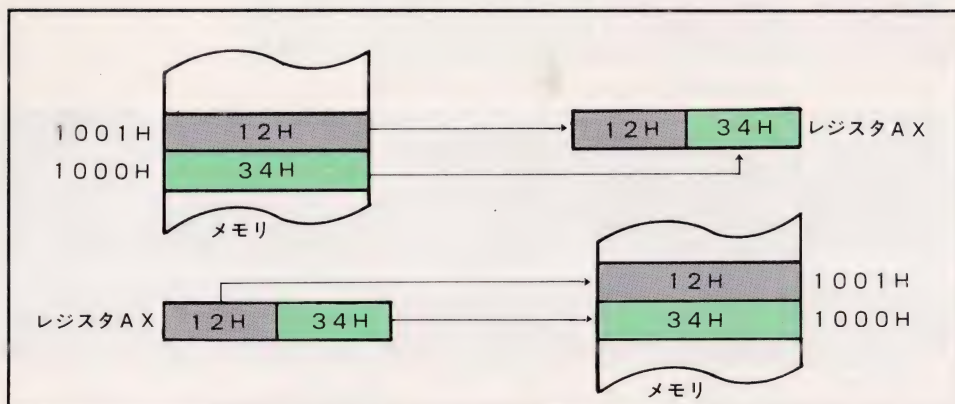
この逆ワードは、人間の側から見ると慣れるまでは不自然に思えるかも知れませんが、良く考えると実に自然な考え方です。なぜならば、逆ワードによってメモリ上に置かれたデータは、上位バイトが上位アドレスのメモリに、下位バイトが下位アドレスのメモリに対応しているからです。

逆ワードが不自然に思えるのは、人間のメモリ・マップに対する考え方に問題があるからで、メモリ・マップを作図する場合にはほとんどの場合に下左図のような図式で表現します(本書も含む)。これを、右図のように考え直せば逆ワードも不自然ではなくなるでしょう。

メモリ・マップの作図



私達はつい00000H番地が先頭だと考えてしまいますが、上位のFFFFFFH番地が先頭だと考えても何ら不都合はないはずです。先ほどの例でためてみましょう。



前おきが長くなりましたが、さっそく16ビットの加減算を実行しましょう。次図のとおりに入力すればプログラムが実行できますが、データをセットする時およびに演算結果を確認する場合、逆ワードであることを注意してください。

```

h]c1800 ← 扱うセグメントを18000H-27FFFH番地に設定する
h]a0000
0000 2E      CS:
0001 A10010  mov ax,[1000]
0004 2E      CS:
0005 03060210 add ax,[1002]
0009 2E      CS:
000A A30410  mov [1004],ax
000D 2E      CS:
000E A10010  mov ax,[1000]
0011 2E      CS:
0012 2B060210 sub ax,[1002]
0016 2E      CS:
0017 A30610  mov [1006],ax
001A F4      hlt
001B
h]s1000
1000 00-0f 00-10 00-00 00-05 00-00 00-00 00-00 00-00
h]d1000,1007
1000 0F 10 00 05 00 00 00 00
h]g0000,001a
*1800:001A

```

プログラムの入力方法と実行方法:

- 扱うセグメントを18000H-27FFFH番地に設定する:** h]c1800
- プログラムを入力する:** h]a0000
- データをセットする:** h]d1000,1007
- プログラムを実行する:** h]g0000,001a
- 最後に:** *1800:001A

hJd1000,1007

1000 0F 10 00 05 0F 15 0F 08

hJ

演算結果を確認する

差

和

演算項

被演算項

3・4・1

CS :

AX←[1001H][1000H]

CS :

AX←AX+[1003H][1002H]

CS :

[1005H][1004H]←AX

CS :

AX←[1001H][1000H]

CS :

AX←AX-[1003H][1002H]

CS :

[1007H][1006H]←AX

HLT

このプログラムではレジスタを1個(AX)しか使っていませんが、他の汎用レジスタを紹介する意味もあって、4個の16ビット汎用レジスタ(AX、BX、CX、DX)を全て使用するように書き換えました。

0000 2E	CS:		1000H~1003H番地に格納する被演算項を尋ねる
0001 A10010	MOV	AX,[1000]	
0004 89C1	MOV	CX,AX	レジスタCXにコピーする
0006 2E	CS:		1002H~1003H番地に格納する演算項を尋ねる
0007 8B1E0210	MOV	BX,[1002]	
000B 89DA	MOV	DX,BX	レジスタDXにコピーする
000D 01D8	ADD	AX,BX	加算を行なう
000F 29D1	SUB	CX,DX	減算を行なう
0011 2E	CS:		和を1004H~1005H番地に格納する
0012 A30410	MOV	[1004],AX	
0015 2E	CS:		差を1006H~1007H番地に格納する
0016 890E0610	MOV	[1006],CX	
001A F4	HLT		ホールド

もちろん、実行方法および実行結果は先ほどの場合と同様です。

```

hJc1800
hJa0000
0000 2E      CS:
0001 A10010  mov ax,[1000]
0004 89C1    mov cx,ax
0006 2E      CS:
0007 8B1E0210 mov bx,[1002]
000B 89DA    mov dx,bx
000D 01D8    add ax,bx
000F 29D1    sub cx,dx
0011 2E      CS:
0012 A30410  mov [1004],ax
0015 2E      CS:
0016 890E0610 mov [1006],cx
001A F4      hlt
001B
hJs1000
1000 0F-0f 10-10 00-00 05-05 0F-00 15-00 0F-00 0B-00
hJd1000,1007
1000 0F 10 00 05 00 00 00 00
hJg0000,001a
*1800:001A
hJd1000,1007
1000 0F 10 00 05 0F 15 0F 0B

```

扱うセグメントを18000H~27FFFH番地に設定する

プログラムを入力する

最後はキーのみでアセンブラから抜ける

データをセットする

データを確認する

プログラムを実行する

演算結果を確認する

差
和
演算項
被演算項

データ（被演算項および演算項）を各レジスタに得た後、

ADD AX, BX

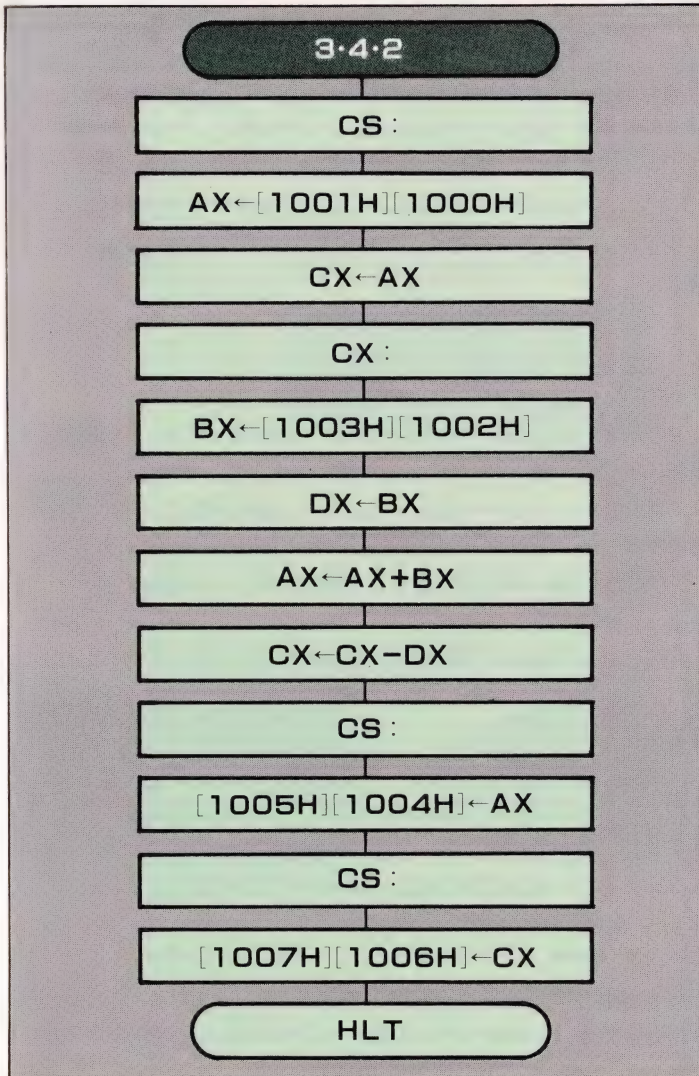
SUB CX, DX

によって演算を行いますが、これらの命令は次の意味を持ちます。

レジスタAX ← レジスタAX + レジスタBX

レジスタCX ← レジスタCX - レジスタDX

レジスタの組み合わせは、ほんの一例で、汎用レジスタの他にもインデックス・レジスタ（SI、DI）やポインタ（BP）なども組み合わせることができます。



3.5 算術演算と論理演算

算術演算

コンピュータ以外の分野において演算と言えば、一般的には算術演算を指すことが多いようです。算術演算と言うと何かむずかしいもののように思えますが、私達が数学(算術)などで普通に使ってきた加減乗除の四則演算が最も代表的な算術演算と言えましょう。

論理演算(ブール代数)

しかし、コンピュータの世界では、新しく論理演算(ブール代数)の概念が導入され、上記の算術演算と同様に幅をきかせています。

AND(論理積)

OR(論理和)

XOR(排他的論理和)

NOT(否定)

EQV(同値)

IMP(包含)

論理演算で代表的なものは、AND(論理積)、OR(論理和)、XOR(E X-OR=排他的論理和)、そしてNOT(否定)です。他にもEQV(同値)やIMP(包含)などがありますが、アセンブリ言語で使用されることはほとんどなく、覚えなくても良いぐらいです。

また、論理演算は2進数から発生した考え方であるため、扱うデータを0と1のみに限定しています。すなわち、通常の10進数や16進数を扱う場合には一度2進数に変換し、ビット(桁)ごとに論理演算を行ないます。

ここで、代表的な4種の論理演算について、それぞれの意味を説明しておきましょう。

●NOT

論理演算の中では、このNOTのみが単項演算子です。単項演算子とは加減乗除のように2つのデータ間で演算を行なうのではなく、データ(演算項)が1つしか必要ないものです。算術演算では負数の直前に付けるマイナス記号(−)等が、この単項演算子にあたります。

NOTは、演算項の上部に補集合を表わすのと同じ横線を引いて、 $\bar{X}$ のように表現します。

左図は真理表と言い、論理演算の演算項(この場合はX)と演算結果(この場合は $\bar{X}$)の対応を表わす場合には、この真理表を用いるとわかりやすく便利です。

真理表を見ればわかるように、NOTは0を1に、1を0に反転するための演算子です。つまり、16進数の00Hに対してこの演算を行なえば結果としてFFHが得られ、55H(01010101B)であればAAH(10101010H)が得られることになります。

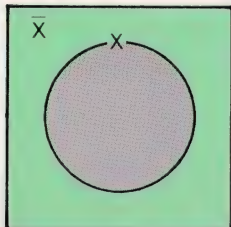
また、論理演算の概念は、集合等で用いるベン図を利用すると考えやすいようです。

NOTの演算結果は常に演算項を否定するため、NOTのXにおける演算結果 $\bar{X}$ は、補集合(円の外側)としてベン図中に表現することができます。

真理表

X	$\bar{X}$
0	1
1	0

ベン図

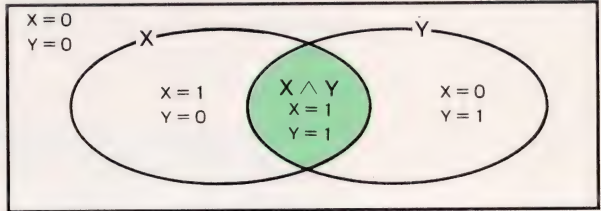


AND

ANDでは2つのデータが両方とも1であった場合にのみ演算結果に1を与え、どちらか一方でも0であれば結果も0になってしまいます。XとYの間でANDを求めることは、 $X \wedge Y$ などと表現します。

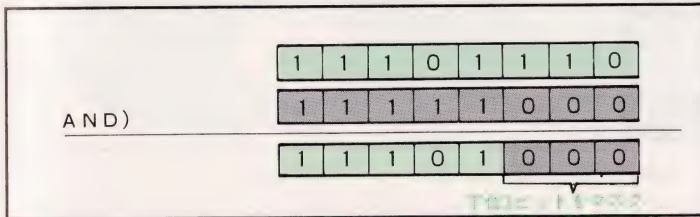
次に真理表とベン図を示します。

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1



一般にANDは、特定のビットをマスクする（強制的に0にする）ために使用できます。たとえば、あるバイト・データの低位3ビットのみを0にマスクしたい場合には、F8H (11111000B)との間でANDを求めます。ために、EEH (11101110B)の低位3ビットをマスクしてみましょう。

マスク



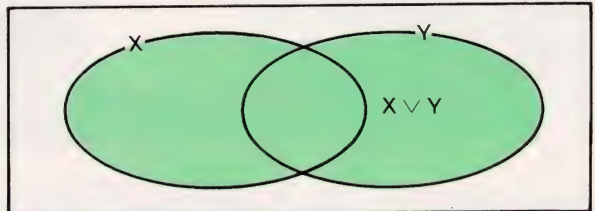
EEH
AND) F8H
E8H

OR

ORでは2つのデータの中の少なくとも片方が1であれば演算結果にも1を与えます。つまり、ORの演算結果が0になるのは、全てのデータ(演算項)が0の場合のみです。

XとYのORであれば、 $X \vee Y$ などと表現します。

X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

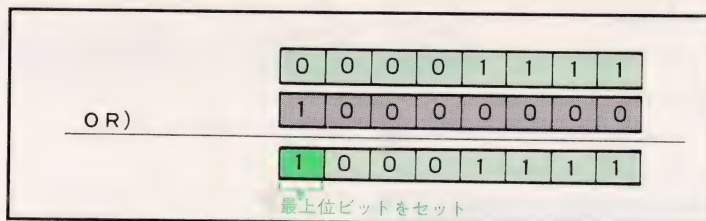


ORはANDとは逆の使い方ができ、特定のビットを強制的に1にセットすることができます。たとえば、あるバイト・データの第7ビットを強制的に1にセットしたい場合には、最上位ビットのみを立てた80H (10000000B)との間でORを求めます。ために0FH (00001111B)の最上位ビットをセットしてみましょう。


```

      0FH
OR)  80H
      8FH

```



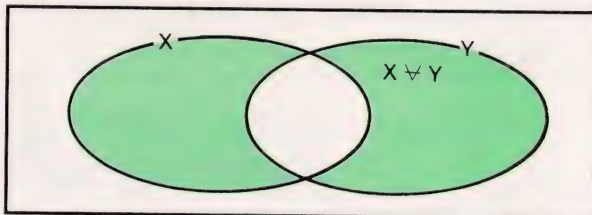
ORという単語は、オペレーション・リサーチ (OPERATION RE-SEARCH) の略語としても使用されます。両者ともコンピュータ関係の文献に登場することが多いので混同しないように注意してください。

●XOR

XOR (EX-OR=イクスクルーシブ・オア) では2つのデータが等しい場合 (0と0または1と1) には演算結果に0を与え、異なる場合 (片方が0でもう一方が1) には結果が1になります。

表現は、ORの場合の∨に横線を入れて、 $X \nabla Y$ などと記します。

X	Y	$X \nabla Y$
0	0	0
0	1	1
1	0	1
1	1	0



XORは、特定のビットをシーソー・スイッチ (フリップ・フロップ) のように使いたい時、すなわち特定のビットを一度変更しておいて、あとで元にもどす必要があるような場合に便利です。

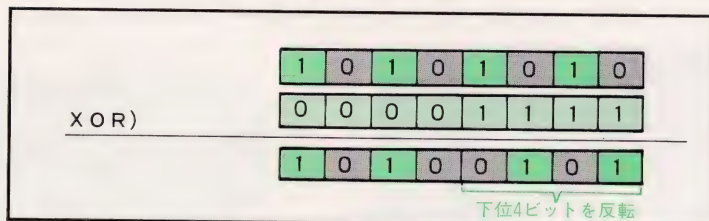
例として、8ビットのデータAAH (10101010B) の下位4ビットを反転して、さらに元にもどすことを考えてみましょう。

まず、XORで下位4ビットを反転します。

```

      AAH
XOR)  0FH
      A5H

```



次に、再度XORを使い、反転した下位4ビットをもとにもどしますが、ここで注目して欲しいのがXORを行なう演算項がどちらの場合にも0FH (00001111B) であることです。この性質を上手に利用すれば、1つのデータと1つのルーチンでビット・パターンのセットとリセットを両方とも行なうことができます。

	1	0	1	0	0	1	0	1
XOR)	0	0	0	0	1	1	1	1
	1	0	1	0	1	0	1	0

	A5H
XOR)	0FH
	AAH

ビット・パターンのセットおよびリセットもためてみましょう。

ビット・パターン00111100Bをセット								
	0	0	0	0	0	0	0	0
XOR)	0	0	1	1	1	1	0	0
	0	0	1	1	1	1	0	0

	00H
XOR)	3CH
	3CH

ビット・パターン00111100Bをリセット								
	0	0	1	1	1	1	0	0
XOR)	0	0	1	1	1	1	0	0
	0	0	0	0	0	0	0	0

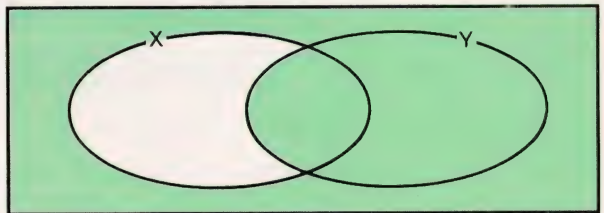
	3CH
XOR)	3CH
	00H

これらの論理演算（ブール代数）にも、私達が数学で学んだ様な定理や法則などが存在します（交換法則、結合法則、分配法則、ド・モルガンの法則など）。また、各演算子の間には優先順位が決められています。

以下に、IMPおよびEQVの真理表のみを示しておきます。

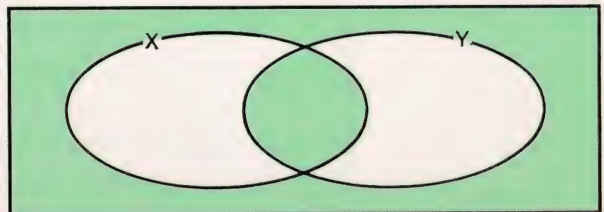
◎ IMP

X	Y	X → Y
0	1	1
0	0	1
1	1	1
1	0	0



◎ EQV

X	Y	X ↔ Y
0	0	1
0	1	0
1	0	0
1	1	1



論理演算の優先順位

NOT
AND
OR
XOR
IMP
EQV

3.6 論理演算をテストするプログラム

AND

OR

XOR

TEST

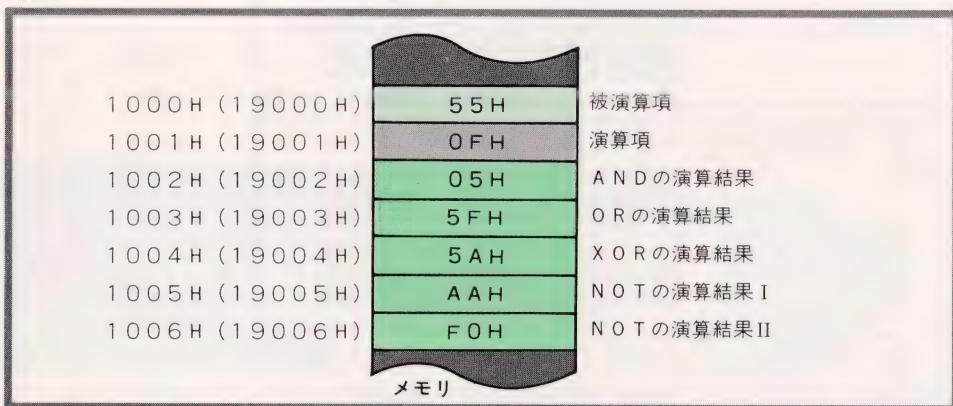
●演算自体はメモリと同等だが演算結果は消滅せず、フラグ(F)のみが影響を受ける命令。

NOT

8086には2項の論理演算命令としてAND、OR、XOR、TESTの4種が、単項のものではNOTが用意されています。

本項では、これらの中でTESTを除いた4種の命令を使って、3.5節で説明した論理演算を実際に行ない、説明が本当に正しかったかどうかを確認してみましょう。

データ・エリアはコード・セグメント中に次図のように設定しておきました。



今回は、被演算項を55H (01010101B) とし、演算項を0FH (00001111B) として、両者の間でAND、OR、XORを求め、さらに両方のNOTを求めることにしました。皆さんも、いろいろな数値を設定した上でプログラムを実行してみてください。きっと論理演算に慣れ親しむことができると思います。

なお、メモリ・マップ中に記載してある数値データは、78ページに示すように、プログラムの実行によってではなく筆算で求めた演算結果で、あくまでも仮定です。

ここで、プログラムによって行なう演算およびデータ・エリアの関係を、もう一度、表にまとめて示します。なお、表中のアドレスは、全てオフセット・アドレスです。

```
[1002H]←[1000H] ^ [1001H].....AND
[1003H]←[1000H] ∨ [1001H].....OR
[1004H]←[1000H] ⊙ [1001H].....XOR
[1005H]←[1000H].....NOT I
[1006H]←[1001H].....NOT II
```

AND

	0	1	0	1	0	1	0	1
AND)	0	0	0	0	1	1	1	1
	0	0	0	0	0	1	0	1

	55H
AND)	0FH
	05H

OR

	0	1	0	1	0	1	0	1
OR)	0	0	0	0	1	1	1	1
	0	1	0	1	1	1	1	1

	55H
OR)	0FH
	5FH

XOR

	0	1	0	1	0	1	0	1
XOR)	0	0	0	0	1	1	1	1
	0	1	0	1	1	0	1	0

	55H
XOR)	0FH
	5AH

NOT I

	0	1	0	1	0	1	0	1
NOT)	1	0	1	0	1	0	1	0

	55H
NOT)	AAH

NOT II

	0	0	0	0	1	1	1	1
NOT)	1	1	1	1	0	0	0	0

	0FH
NOT)	F0H

次ページに私のつくったプログラムのリストを示します。

冒頭で、1000H番地の被演算項をレジスタALに、1001H番地の演算項をレジスタAHに得るために、

```
MOV     AX, [1000H]
```

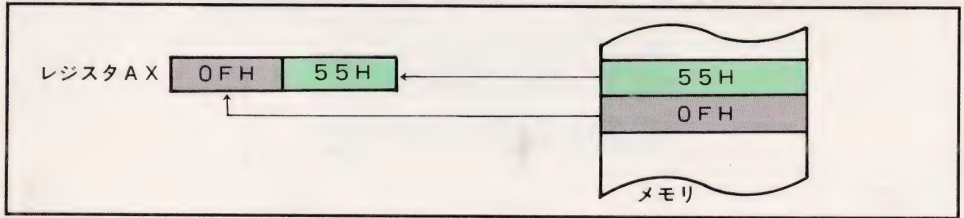
を行っていますが、プログラムの見易さを優先したければ、

```
MOV     AL, [1000H]
```

```
MOV     AH, [1001H]
```

と、2ステップに分割しても全くさしつかえありません。ここでは、レジスタAX、レジスタAH、レジスタALの関係をしっかりと理解する目的もあったため、前者の手法を用いました。

当然、このような場合にも逆ワードとなることに注意してください。



このため、たとえばANDを求める命令も、

AND AH, AL

ではなく、

AND AL, AH

- 被演算項と演算項を区別する2項演算
⇒減算、除算、IMP等
- 被演算項と演算項を区別しない2項演算
⇒加算、乗算、AND、OR、XOR、ECX等

を使用しています。しかし、8086の2項論理演算 (AND、OR、XOR、TEST) は、いずれも、どちらの項を被演算項に設定しても演算結果は変わりません。X ∧ YでもY ∧ Xでも結果は等価です。

ただし演算すべてが被演算項と演算項を区別しないわけではありませんから注意してください。減算が良い例で、X - YとY - Xが必ずしも等しくないことは一目でわかると思います。

0000 2E	CS:			
0001 A10010	MOV	AX, [1000]		被演算項をレジスタAとし、演算項をレジスタHに格納する
0004 89C3	MOV	BX, AX		レジスタBとレジスタHにコピーする
0006 89C1	MOV	CX, AX		レジスタCとレジスタHにコピーする
0008 89C2	MOV	DX, AX		レジスタDとレジスタHにコピーする
000A 20E0	AND	AL, AH		
000C 2E	CS:			ANDを求め格納する
000D A20210	MOV	[1002], AL		
0010 08FB	OR	BL, BH		ORを求め格納する
0012 2E	CS:			
0013 881E0310	MOV	[1003], BL		
0017 30E9	XOR	CL, CH		XORを求め格納する
0019 2E	CS:			
001A 880E0410	MOV	[1004], CL		
001E F7D2	NOT	DX		NOTを求め格納する
0020 2E	CS:			
0021 89160510	MOV	[1005], DX		
0025 F4	HLT			ホールド

次のページから3ページにわたって、順にフローチャート、データの流れ、プログラムの実行手順を示します。

皆さんも、いろいろなデータを設定してこのプログラムを実行し、論理演算結果を確認してみてください。

3.6

CS :

AX←[1001H][1000H]

BX←AX

CX←AX

DX←AX

AH←AH^AL

CS :

[1002H]←AH

BH←BH∨BL

CS :

[1003H]←BH

CH←CH∨CL

CS :

[1004H]←CH

DX←DX

CS :

[1006H][1005H]←DX

HLT

データの流れ

[1000H]

1001H

レジスタAL

レジスタAH

AND

レジスタAL

[1002H]

レジスタBL

レジスタBH

OR

レジスタBL

[1003H]

レジスタCL

レジスタCH

XOR

レジスタCL

[1004H]

レジスタDL

レジスタDH

NOT

レジスタDL

[1005H]

レジスタDH

[1006H]


```

h]c1800
h]a0000
0000 2E      cs:
0001 A10010  mov ax,[1000]
0004 89C3    mov bx,ax
0006 89C1    mov cx,ax
0008 89C2    mov dx,ax
000A 20E0    and al,ah
000C 2E      cs:
000D A20210  mov [1002],al
0010 08FB    or bl,bh
0012 2E      cs:
0013 881E0310 mov [1003],bl
0017 30E9    xor cl,ch
0019 2E      cs:
001A 880E0410 mov [1004],cl
001E F7D2    not dx
0020 2E      cs:
0021 89160510 mov [1005],dx
0025 F4      hlt
0026

```

セグメントを指定する

プログラム・リストを入力する

```

h]s1000
1000 00-55 00-0f 00-00 00-00 00-00 00-00 00-00
h]d1000,1006
1000 55 0F 00 00 00 00 00
h]g0000,0025
*1800:0025
h]d1000,1006
1000 55 0F 05 5F 5A AA F0
h]s1000
1000 55-f0 0F-0f 05-00 5F-00 5A-00 AA-00 F0-00
h]d1000,1006
1000 F0 0F 00 00 00 00 00
h]g0000,0025
*1800:0025
h]d1000,1006
1000 F0 0F 00 FF FF 0F F0
h]

```

データをセットする

データを確認する

プログラムを実行する

演算結果を確認する

他のデータでためしてみる

NOT II

NOT I

XOR

OR

AND

演算項

被演算項

3.7 16ビットのデータを10倍するプログラム

アセンブリ言語で簡単な乗算を行なってみましょう。

8086には乗除算のための命令が用意されていますが、ここでは加算命令の組み合わせによって10倍することを考えてみました。

Xの値を10倍する式、「 $X \leftarrow X * 10$ 」を次のように変形します。

$$X \leftarrow ((X * 2) * 2 + X) * 2$$

$$X * 2 = X + X$$

ここで、「 $X * 2$ 」は、「 $X + X$ 」と等価ですから、加算命令のみで何とか間に合いそうです。

プログラムでは、コード・セグメント中の1000H番地から1001H番地に渡る16ビットのデータを10倍して、同じアドレスのメモリに格納するようにしました。

1000H (19000H)

0AH

データの低位8ビット

1001H (19001H)

00H

データの高位8ビット

実行前

1000H (19000H)

64H

演算結果の低位8ビット

1001H (19001H)

00H

演算結果の高位8ビット

実行後

$$000AH = 10$$

$$0064H = 100$$

$$03E8H = 1000$$

上記のメモリ・マップは、データ000AH (10進数では10) を10倍した結果として0064H (100) が得られる様子を表現したものです。

さっそくプログラム・リストおよびフローチャートを示します。

```
0000 2E
0001 A10010
0004 01C0
0006 01C0
0008 2E
0009 03060010
000D 01C0
000F 2E
0010 A30010
0013 F4
```

```
CS:
MOV    AX,[1000]
ADD    AX,AX
ADD    AX,AX
CS:
ADD    AX,[1000]
ADD    AX,AX
CS:
MOV    [1000],AX
HLT
```

データを2倍する

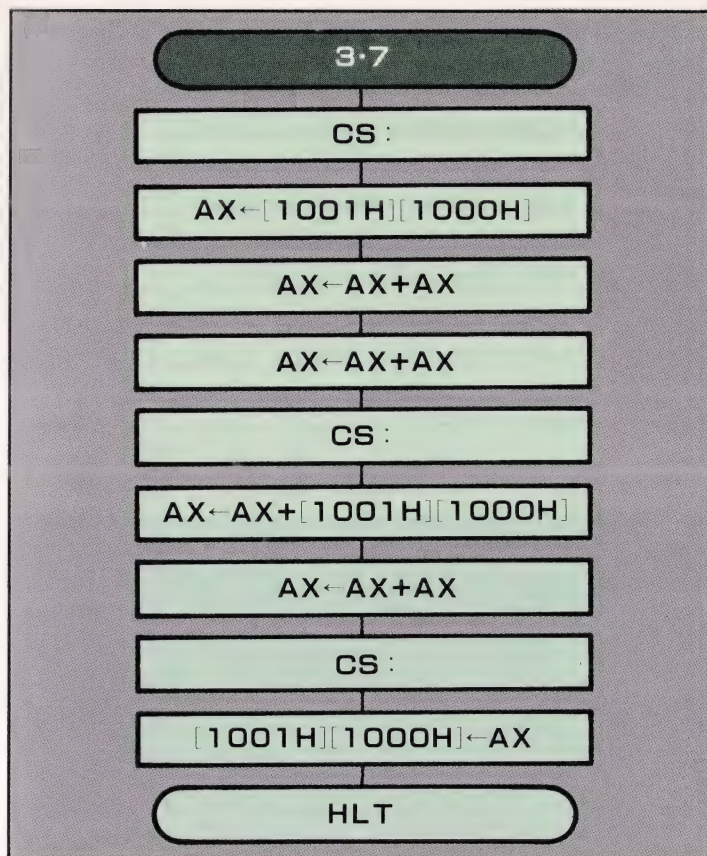
データを4倍する

データを5倍する

データを10倍する

演算結果を格納する

ホールド



今回は10倍することを考えてプログラムをつくりましたが、どちらかの演算項が決まっている（定数）乗算を行なう場合には、乗算命令を用いなくても比較的簡単に演算をおこなうことができます。

もう1つの例として、レジスタAXのデータを160倍することを考えてみましょう。

加算命令を続けて実行して行くことによって、2倍、4倍、8倍、16倍、32倍……と、2のべき乗を乗じた値が求められるのは容易に想像がつくと思います。そこで、レジスタAXのデータを160倍するための式「 $AX \leftarrow AX * 160$ 」を、2のべき乗を乗じた値の和によって表現できるように変形します。

$$AX \leftarrow (AX * 32) + (AX * 128)$$

この式が求められれば、あとは簡単で、レジスタAXのデータを、2倍、4倍、8倍、16倍、32倍、64倍、128倍と増して行きながら、32倍の値と128倍の値を加えれば、結果としてレジスタAXのデータを160倍したことになります。

2倍
4倍
8倍
16倍
32倍
64倍
128倍
256倍
512倍
⋮

32倍したものと
128倍したものを
加えると160倍し
たことになります

以上の理論によってコーディングしたサンプルを示します。

ADD	AX, AX	レジスタAXのデータを2倍する
ADD	AX, AX	レジスタAXのデータを2倍する (4倍)
ADD	AX, AX	レジスタAXのデータを2倍する (8倍)
ADD	AX, AX	レジスタAXのデータを2倍する (16倍)
ADD	AX, AX	レジスタAXのデータを2倍する (32倍)
MOV	BX, AX	32倍した値をレジスタBXに退避しておく
ADD	AX, AX	レジスタAXのデータを2倍する (64倍)
ADD	AX, AX	レジスタAXのデータを2倍する (128倍)
ADD	AX, BX	128倍した値に退避しておいた32倍の値を加える

次図は、10倍するプログラムの実行例です。

h]c1800	←	セグメントを指定する
h]a0000		
0000 2E	CS:	
0001 A10010	mov ax,[1000]	
0004 01C0	add ax,ax	
0006 01C0	add ax,ax	
0008 2E	CS:	
0009 03060010	add ax,[1000]	プログラム・リストを入力する
000D 01C0	add ax,ax	
000F 2E	CS:	
0010 A30010	mov [1000],ax	
0013 F4	hlt	
0014		
h]s1000		
1000 00-0a 00-00		データをセットする
h]g0000,0013		
*1800:0013		プログラムを実行する
h]d1000,1001		
1000 64 00		
h]■		演算結果を確認する

3.8 サブルーチンの使い方

前節(3.7)の10倍するプログラムをもとにして、100倍するプログラムをつくってみました。

100倍するということは、10倍したものをさらに10倍することですから、プログラムは次のようになります。

$$100 = 10 * 10$$

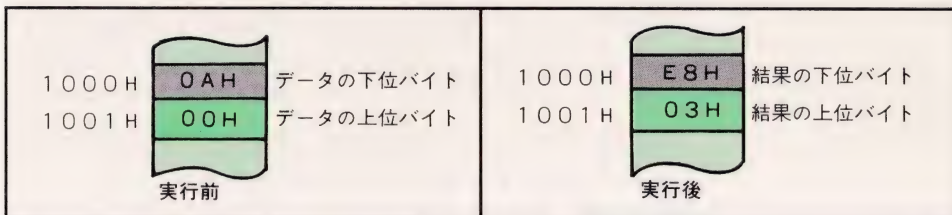
0000 2E	CS:		
0001 A10010	MOV	AX,[1000]	
0004 01C0	ADD	AX,AX	
0006 01C0	ADD	AX,AX	
0008 2E	CS:		
0009 03060010	ADD	AX,[1000]	
000D 01C0	ADD	AX,AX	
000F 2E	CS:		
0010 A30010	MOV	[1000],AX	
0013 2E	CS:		
0014 A10010	MOV	AX,[1000]	
0017 01C0	ADD	AX,AX	
0019 01C0	ADD	AX,AX	
001B 2E	CS:		
001C 03060010	ADD	AX,[1000]	
0020 01C0	ADD	AX,AX	
0022 2E	CS:		
0023 A30010	MOV	[1000],AX	
0026 F4	HLT		

カレント・コード・セグメント中1000H番地から1001H番地に渡るデータを10倍する

さらに10倍する(結果的には100倍)

ホールド

10倍する方法(アルゴリズム)に関しては前節(3.7)を御参照ください。データは、1000H番地(19000H番地)から2バイトに渡っている16ビットの数値で、プログラムを実行することによって、データを100倍した値が同じアドレスに格納されます。



今までどおり、データは逆ワードで与えますし、演算結果も逆ワードで得られます。次のページにプログラムの実行方法を示しますが、私の場合には、000AH(10進数では10)を100倍してみました。


```

hJc1800 ←
hJa0000
0000 2E      CS:
0001 A10010  mov ax,[1000]
0004 01C0    add ax,ax
0006 01C0    add ax,ax
0008 2E      CS:
0009 03060010 add ax,[1000]
000B 01C0    add ax,ax
000F 2E      CS:
0010 A30010  mov [1000],ax
0013 2E      CS:
0014 A10010  mov ax,[1000]
0017 01C0    add ax,ax
0019 01C0    add ax,ax
001B 2E      CS:
001C 03060010 add ax,[1000]
0020 01C0    add ax,ax
0022 2E      CS:
0023 A30010  mov [1000],ax
0026 F4      hlt
0027
hJs1000
1000 64-0a 00-00
hJg000,0026
*1800:0026
hJd1000,1001
1000 E8 03
hJ■

```

セグメントを指定する

プログラム・リストを入力する

データをセットする

プログラムを実行する



演算結果を確認する

ところで、この100倍するプログラムのリストをながめていて、何か気がつくことはありませんか？

注意深く見てみると、全く同じ部分が2か所あります。当然のことかも知れませんが、10倍するルーチンを、そのまま2回使っているのです。

この10倍ルーチンを1か所分だけ用意しておいて、2か所から呼び出して使おうと考えて組み直したものが、次のプログラムです。

◎メイン・ルーチン

0000 E80400	CALL 0007	10倍するサブルーチンと呼ぶ
0003 E80100	CALL 0007	さらに10倍するサブルーチンと呼ぶ
0006 F4	HLT	ホールド

◎ サブルーチン

0007 2E	CS:		
0008 A10010	MOV	AX, [1000]	
0008 01C0	ADD	AX, AX	
000D 01C0	ADD	AX, AX	
000F 2E	CS:		10倍するサブルーチン
0010 03060010	ADD	AX, [1000]	
0014 01C0	ADD	AX, AX	
0016 2E	CS:		
0017 A30010	MOV	[1000], AX	
001A C3	RET		メイン・ルーチンへもどる

上記のプログラムは、メイン・ルーチンとサブルーチンに分離することができ、メイン・ルーチンからサブルーチン呼び出すために、サブルーチン・コール命令 (CALL) を使用しています。

CALL
RET

ここでは、サブルーチンが0007H番地からに位置しているため、
CALL 0007H
を用いて0007H番地をコールしており、サブルーチンの最後 (001AH番地) に置かれたリターン命令 (RET) でメイン・ルーチンにもどっています。

この例でもわかるように、サブルーチン・コールの一般的な書式は、

CALL (サブルーチンの開始アドレス)

であり、サブルーチンからメイン・ルーチンにもどるための命令は、

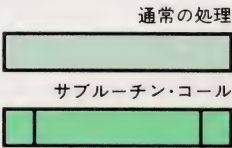
RET

です。

サブルーチン化

サブルーチン化を行なうことによって、プログラムは見やすくわかりやすくなりますし、使用するメモリの容量も少なくて済みます。また、プログラムの改造や改良が必要な場合にも、一部の修正が全体に波及するようなことを防ぐことができます。このような見地からも、プログラムの中にサブルーチン化できる可能性が存在する場合には、できる限りサブルーチンを用いた方が良いことがわかりますし、実際にアセンブリ言語のプログラムでは、非常に多くのサブルーチンを使うのが普通です。

次ページに、100倍するプログラムをサブルーチン化する前のフローチャート (左側)、およびサブルーチン化後のフローチャート (右側) を示します。フローチャート中で、サブルーチン・コールの箱は、通常の処理の箱の両端に線を入れた右図の様な形を用い、箱の中部にはコールするサブルーチンの名称、ラベル、または開始アドレス等を記入します。



3.8.1

CS:

AX ← [1001H][1000H]

AX ← AX + AX

AX ← AX + AX

CS:

AX ← AX + [1001H][1000H]

AX ← AX + AX

CS:

[1001H][1000H] ← AX

CS:

AX ← [1001H][1000H]

AX ← AX + AX

AX ← AX + AX

CS:

AX ← AX + [1001H][1000H]

AX ← AX + AX

CS:

3.8.2

MLTTEN

MLTTEN

HLT

SUBROUTINES

MLTTEN

CS:

AX ← [1001H][1000H]

AX ← AX + AX

AX ← AX + AX

CS:

AX ← AX + [1001H][1000H]

AX ← AX + AX

CS:

[1001H][1000H] ← AX

RET

[1001H][1000H] ← AX

HLT

コール命令 (CALL) によってサブルーチンを呼び出し、リターン命令 (RET) でメイン・ルーチンにもどることは、容易に理解していただけたと思います。

ところで、このサブルーチン・コールにおいて、8086は実際にはどのような働きを行なっているのでしょうか？ CPUの立場に立って少し考えてみましょう。

まず、サブルーチン・コール時にCPUが行なうことは、サブルーチンからもどる時のために、現在の命令ポインタ (IP) の値を退避することですが、退避する場所はスタックと呼ばれる領域です (スタックについては後述)。次に、コール先サブルーチンのアドレスが命令ポインタ (IP) に送り込まれるため、CPUはサブルーチンの先頭から命令をフェッチし、実行を続けます。

リターン命令ではこれと全く逆の動作を行ない、スタックに退避してあった、もどり先のアドレスをスタックから取り出して、命令ポインタ (IP) に送り込むことによって、メイン・ルーチンへもどることができます。

最後になりましたが、プログラムの実行手順を示しておきます。

CPUから見た
サブルーチン・コール

スタック

```
h]c1800 ← セグメントを設定する
h]a0000
0000 E80400      call 0007
0003 E80100      call 0007
0006 F4          hlt
0007 2E          cs:
0008 A10010      mov ax,[1000]
000B 01C0        add ax,ax
000D 01C0        add ax,ax
000F 2E          cs:
0010 03060010    add ax,[1000]
0014 01C0        add ax,ax
0016 2E          cs:
0017 A30010      mov [1000],ax
001A C3          ret
001B
h]s1000          データをセットする
1000 64-0a 00-00
h]g0000,0006     プログラムを実行する
*1800:0006
h]d1000,1001
1000 E8 03
h]■
```

演算結果を確認する

3.9 スタックの概念とレジスタ退避

8086には、いくつものレジスタが用意されていますが、ある程度以上の長さのプログラムでは、レジスタを効率良くフルに使用することが必要となります。

このように、全てのレジスタを活用している時、もし少しの間、どうしても別の目的にレジスタを空けたくなったらどうすれば良いでしょうか？ もちろん各レジスタの内容は、全てもと通りにもどしておかなければならないものとします。

たとえば、レジスタAXをあける場合に、

```
MOV      [1000H], AX
```

などで、レジスタAXのデータをメモリ上（この場合には1000H～1001H番地）に退避しておき、レジスタの内容をもとにもどす場合にもMOVを使って次のようにすることが考えられます。

```
MOV      AX, [1000H]
```

しかし、この方法でレジスタの内容を退避しておくのには、いくつかの問題点が存在します。

まず、上記の方法では全ての場合において、退避するメモリのアドレスを指定する必要があるため、レジスタの内容を退避した先のアドレスを全て記憶しておかなければ、後でレジスタに値をもどす場合に困ってしまいます。下手をすると同じアドレスに重ねて何度も退避してしまって気がつかず、デバッグの時に大変な苦勞を強いられることもあり得ます（下図参照）。

MOV	[1000H], AX	レジスタAXを退避する
MOV	[1000H], BX	気づかずにレジスタBXを重ねてしまう
MOV	AX, [1000H]	間違った値がレジスタAXにもどされる

上図の場合には、命令自体が3ステップしかありませんから間違えようがないのですが、たとえば1000ステップ2000ステップもあるプログラムの中に上記の命令が散在していることを考えてください。本格的なアセンブラであれば、データ・エリアのアドレス（この場合には1000H番地）にラベル（名称）がつけられますので、ほとんど問題ありませんが、モニタ内蔵のアセンブラや、対応表をひきながらハンド

・アセンブルを行なう場合には要注意です。さらに、省メモリのことを考えても、アドレス指定のためにメモリを2バイトずつ浪費するのは考えものです。

そこで、メモリへのレジスタ退避をより効率良く行なうために、スタックの考え方が必要になるのです。

それでは、スタックとはいったい何なのでしょう？

スタック

コンピュータでいうスタックとは単にメモリの使い方です。先ほどの例でもわかるように、普通の転送命令 (MOV) で、レジスタとメモリ間のデータ転送を考えれば、そのたびごとに、メモリのアドレスを指定してやらなければなりません。

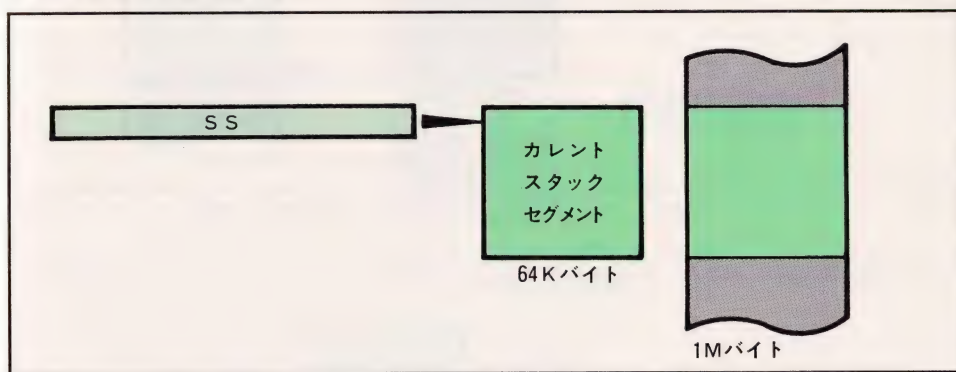
そこで、アドレス指定のために専用のレジスタを用意し、その値を自動的に増減してやれば、転送のたびにアドレスを指定する必要がなくなるわけです。

皆さんも、もうお気づきかも知れませんが、このアドレス指定専用のレジスタがスタック・ポインタ (SP) であり、またスタック・セグメント・レジスタ (SS) であるのです。

スタック・ポインタ
(SP)

そして、スタック・セグメント・レジスタによって先頭アドレス (物理アドレス) を指定される64Kバイトのセグメントを、カレント・スタック・セグメントと呼びます。

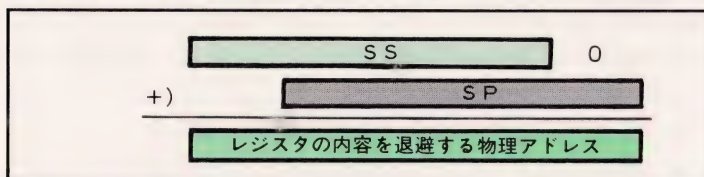
スタック・セグメント・
レジスタ(SS)



ここで、スタック・セグメント・レジスタ (SS) の末尾には、自動的に4ビットの0が付加されることに注意してください。すなわち、スタック・セグメント・レジスタ (SS) に1000Hが入っている場合には、1000H番地からの64Kバイトがカレント・スタック・セグメントとなります。

スタック (スタック・エリア) は、カレント・スタック・セグメントの中に置かなければなりません、具体的にスタックのオフセット・アドレスを指定するのがスタック・ポインタ (SP) です。すなわち、実際にレジスタの内容を退避するメモリのアドレスは、スタック・セグメント

・レジスタ (SS) のセグメント・アドレスに、スタック・ポインタ (SP) のオフセット・アドレスを加えたものです。

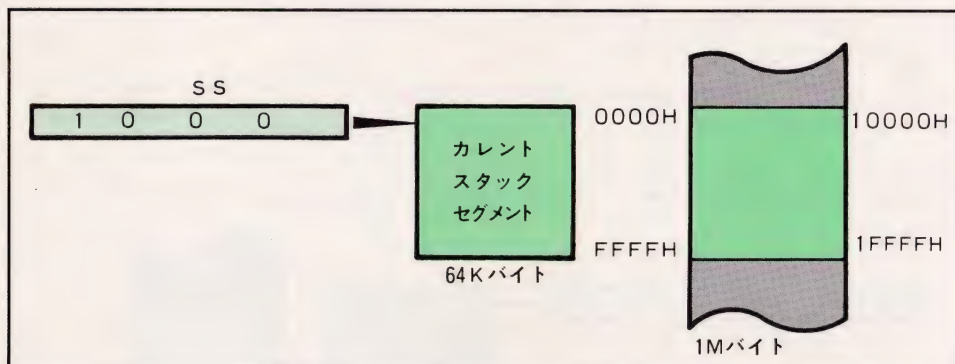


ここで、実際にスタックの構造を説明したいのですが、抽象的な例題だけでは説明がわかりにくくなってしまいますので、仮に、スタック・セグメント・レジスタ (SS) に 1000Hが入っており、スタック・ポインタ (SP) には FFFFHが入っているものとして具体例を中心に話を進めます。

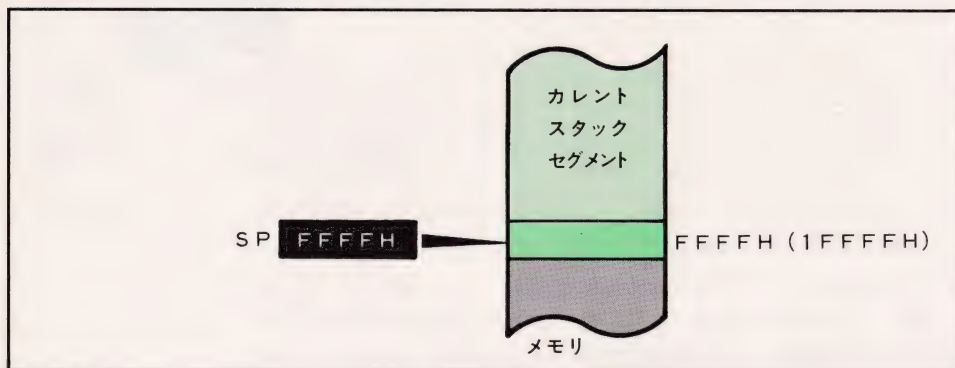
SS ← 1000H

SP ← FFFFH

この状態では、10000H番地から 1FFFFH番地までの64Kバイトがカレント・スタック・セグメントとして設定されています。



また、スタック・ポインタは FFFFH番地(実際には 1FFFFH番地)を指定していることになります。



次に、この状態からレジスタ AXのワード・データをスタックに退避してみましょう。あらかじめレジスタ AXには、1234Hが格納さ

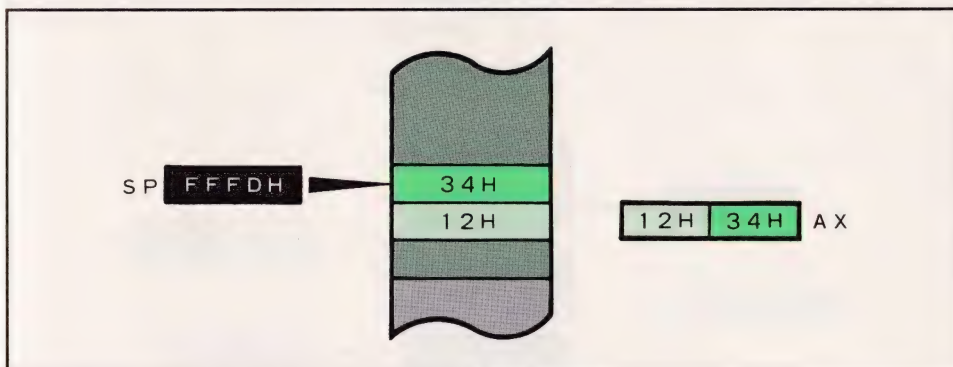
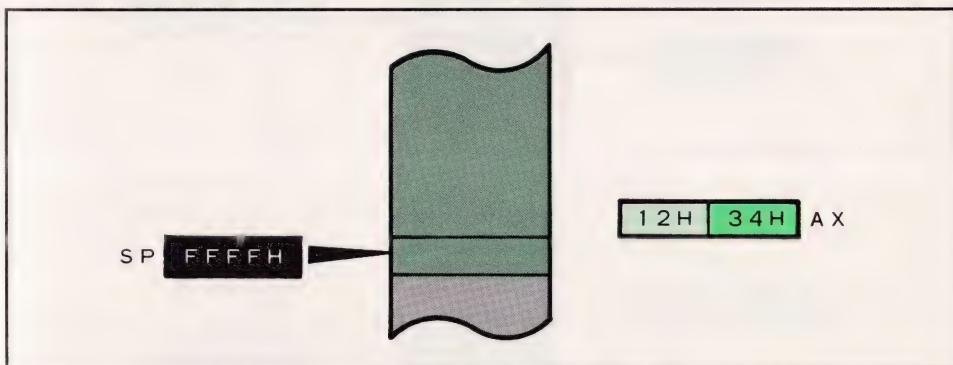
れているものとします。

なお、レジスタAXの内容をスタックに退避する命令はPUSHを用いて次のようにします。

PUSH AX

PUSH

次図はPUSHを用いて、ワード・データ(1234H)をスタックに退避した直前および直後の様子です。



レジスタのワード・データをスタックに退避(PUSH)する際に、CPUが行なう作業を分解すると、つぎのようになります。

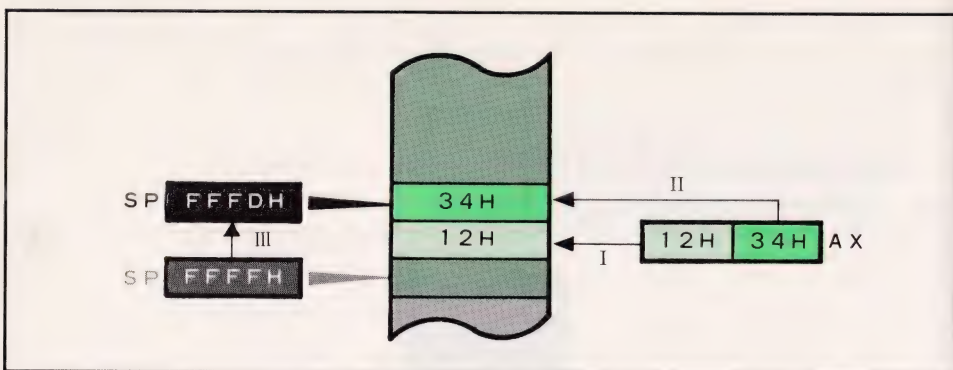
- I (SP-1) 番地にデータの上位バイトを退避する
- II (SP-2) 番地にデータの下位バイトを退避する
- III SPの内容から2を減じる

先ほどの具体例にそって動作を追ってみましょう。

まず、Iによって、(FFFFH-1)番地すなわちFFFEH番地にレジスタAXの上位バイトが退避されます。

次に、IIで、(FFFFH-2)番地すなわちFFFDH番地にレジスタAXの下位バイトが退避されます。

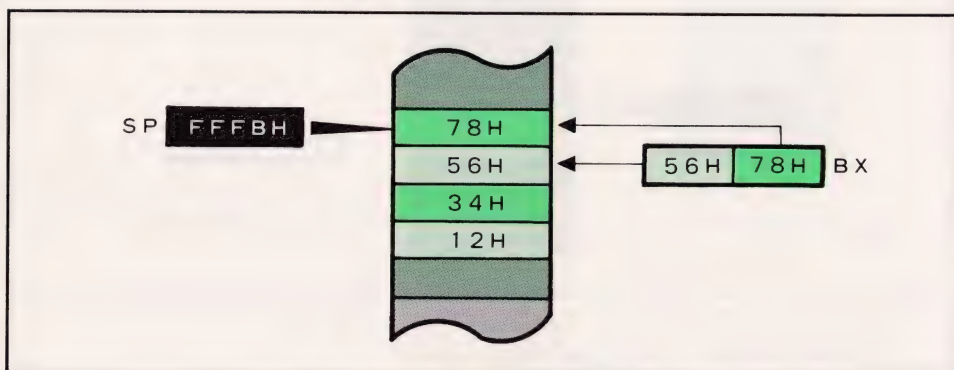
IIIの動作により、FFFFHであったスタック・ポインタ(SP)がFFFDH番地を指定するようになり、レジスタ退避が完了します。



ここで重要なのが、スタック・ポインタ (SP) の値が自動的に2ずつ減じられて行く点です。このため、連続してレジスタ退避を行なうことによって、メモリの若い方へと次々にデータを格納して行くことが可能です。

ために、レジスタBXに5678Hを入れて、それをスタックに退避してみましょう。

```
MOV     BX, 5678H
PUSH    BX
```



今度は、スタック (スタック・エリア) に退避しておいたデータをレジスタBXにもどす場合を考えてみましょう。

POP

```
POP     BX
```

この命令 (POP) は、PUSHとは逆の機能を持った命令で、最後 (直前) にスタックへ退避しておいたワード・データをレジスタに戻すために用いられます。これをレジスタ退避に対して、レジスタ退避解除と言う場合があります。

さっそく、POPを実行する際にCPUが行なう作業を分解してみました。

I レジスタの下位バイトにSP番地のデータを戻す

II レジスタの上位バイトに(SP+1)番地のデータをもどす

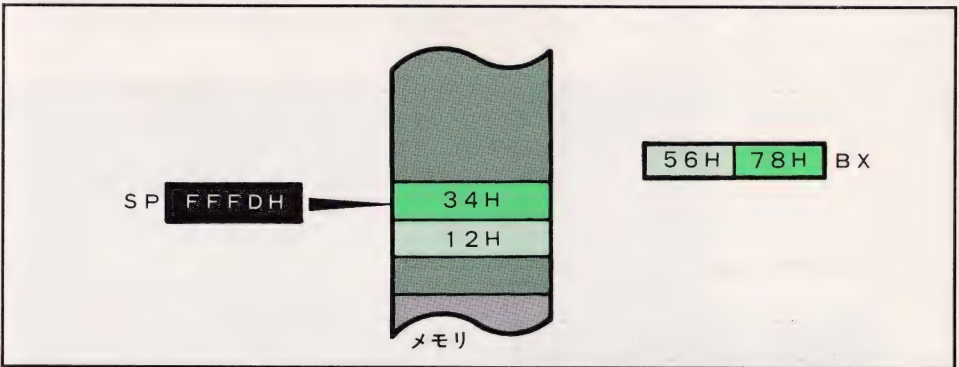
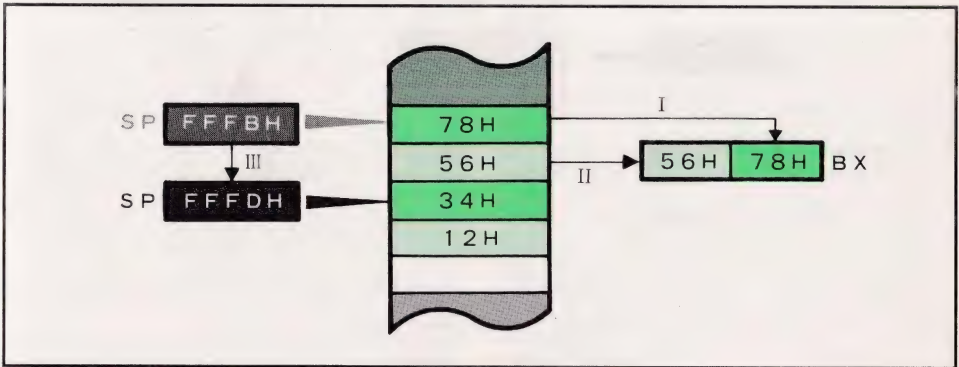
III SPの内容に2を加える

先ほどの例でスタックに退避したワード・データ(5678H)をレジスタBXにもどす場合はどうなるでしょうか？

Iによって、FFFBH番地の内容すなわち78HがレジスタBXの下位バイトにもどされます。

次にIIによって、FFFDH番地の内容すなわち56HがレジスタBXの上位バイトにもどされます。

最後に、FFFDH番地を指定していたスタック・ポインタ(SP)に2が加えられ、FFFDH番地を指定するようになります。

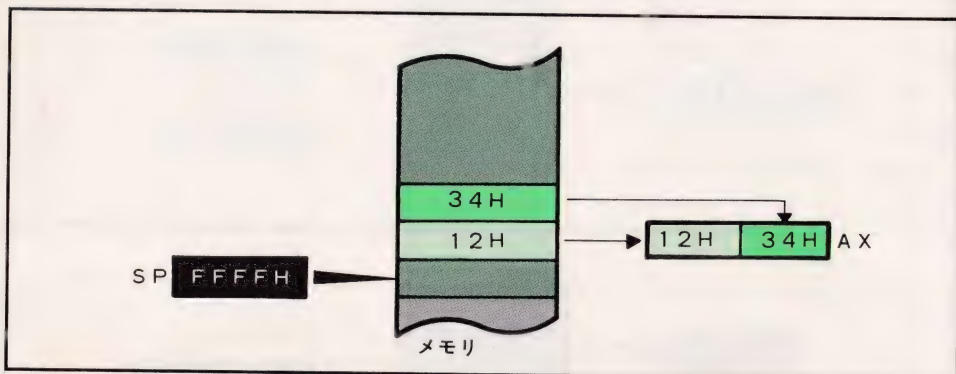


図からもわかるように、連続してPOPを実行することで、メモリの下の方から上位に向かって次々にレジスタにデータをもどして行くことが可能です。

POPの使用にあたって注意する点は、基本的にPUSHの回数とPOPの回数が等しくなければならないことです。ある程度スタックの考え方に慣れれば、この限りではありませんが、初めのうちは、PUSHしないデータはPOPしないこと、PUSHしたデータは必ずどこかでPOPしておくことをおすすめします。

さて、皆さんにすすめたからには私も責任を持って、最初にPUSHしたレジスタAXのデータ(1234H)をもとにもどしておきましょう。

POP AX

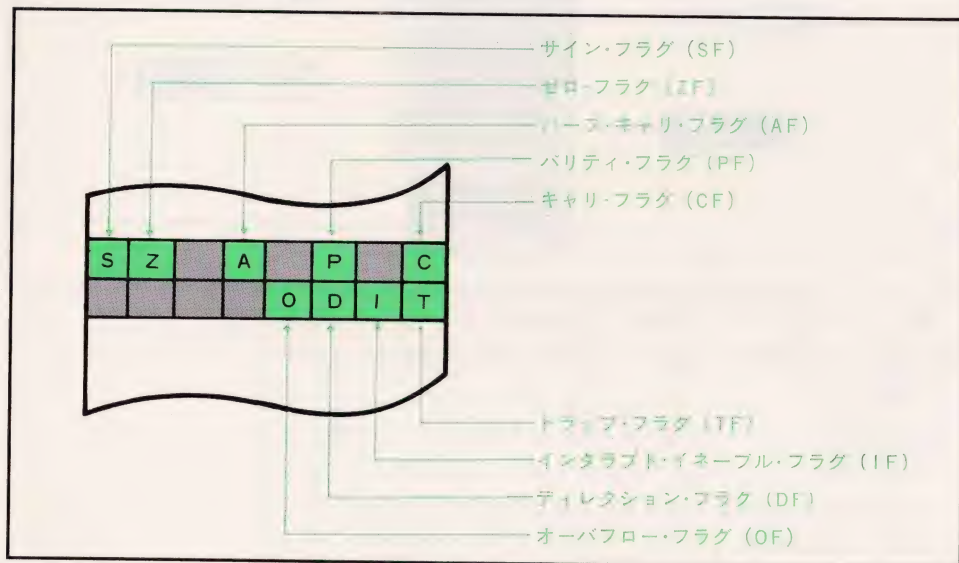


ここまでの説明は、全て汎用レジスタの退避に限って来ましたが、実際にはセグメント・レジスタはもちろんのこと、メモリ上のデータでさえもスタックに退避することができます。

PUSHF
POPF

また、フラグ (F) の内容をPUSHおよびPOPするために特別な命令 (PUSHFおよびPOPF) も用意されています。

PUSHFは、全てのフラグをワード・データとしてスタックに退避する命令で、スタックに退避された内容は次図のような意味を持っています。



POPFはスタックのデータをフラグ (F) にもどす命令です。

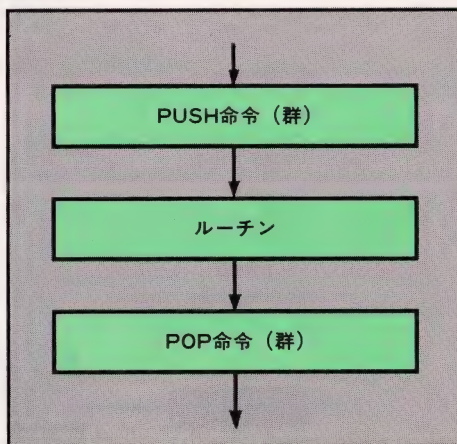
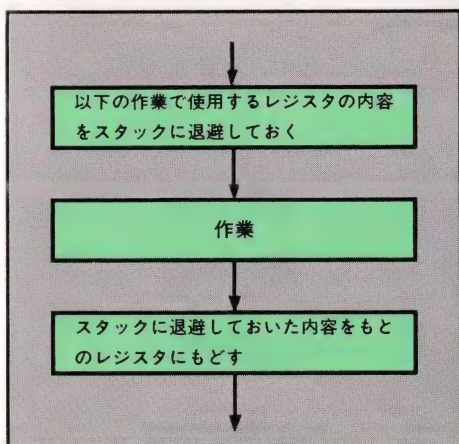
3・10 レジスタ退避の使用方法

レジスタ退避のための命令が、PUSH (PUSHFも含む) とPOP (POP) に分類できることは前述しました。

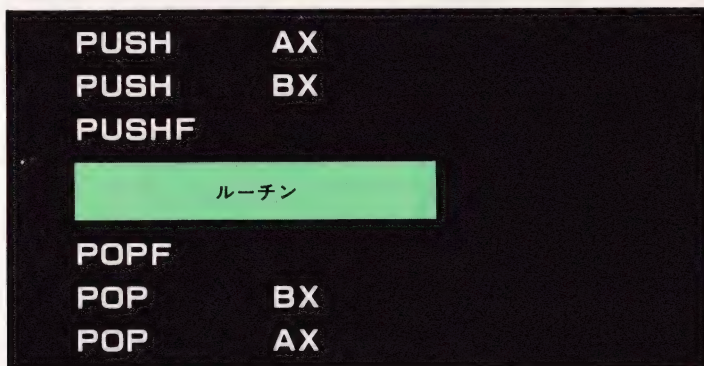
PUSH PUSHF
POP POPF

しかし、このPUSH系の命令とPOP系の命令が連続して使用されることはほとんどありません。

次図に一般的なレジスタ退避の使用方法を示します。



たとえば、レジスタAX、レジスタBXおよびフラグ(F)の内容を退避し、もとにもどす一般的な手順は次のようになります。



ここで、PUSH時とPOP時のレジスタの順番が逆になっている点に注目してください。

実際のプログラム中では、最後(直前)にPUSHしたデータが最初にPOPできるスタックの構造(LAST IN FIRST OUT STRUCTURE)を巧みに利用した多くの手法が使われています。

LAST IN
FIRST OUT

3.11 レジスタ退避の応用例

本節では、スタックの概念に慣れ親しむため、レジスタ退避の応用方法を考えてみましょう。

まず、16ビットのレジスタ間で、データを転送（コピー）する場合、たとえばレジスタBXの内容をレジスタAXに移す場合には、普通MOVを用いて次のようにします。

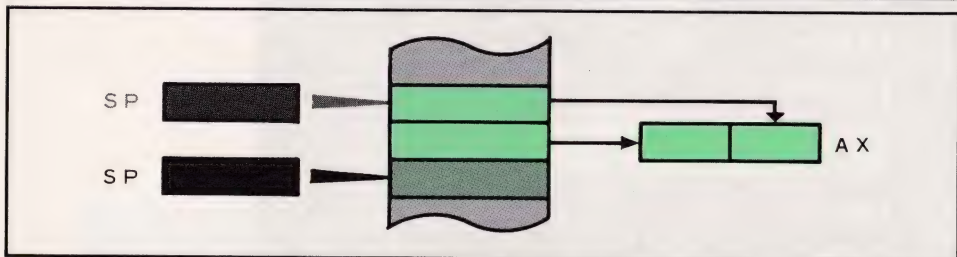
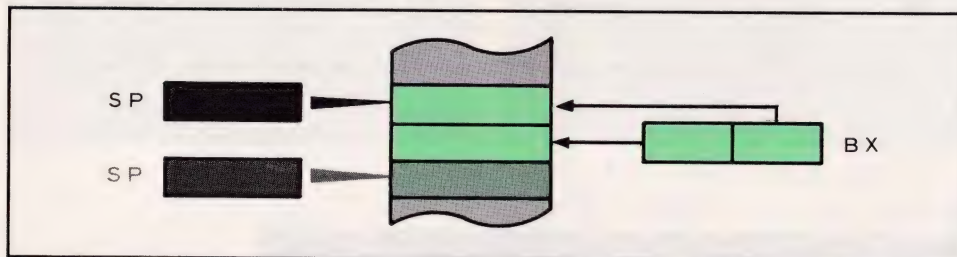
```
MOV      AX, BX
```

同様の動作を行なわせるのに、少しおもしろい方法として、

```
PUSH     BX
```

```
POP      AX
```

を実行することが考えられます。まず、レジスタBXの内容をスタックに退避しておいて、今度はスタックの内容をレジスタAXに引き出せば、結果として、レジスタBXの内容をレジスタAXに転送したのと同じこととなります。



次に、レジスタ退避を利用して、2つのレジスタの内容を交換してみましょう。ソース・インデックス・レジスタ（SI）とデスティネーション・インデックス・レジスタ（DI）の内容を交換する場合には、

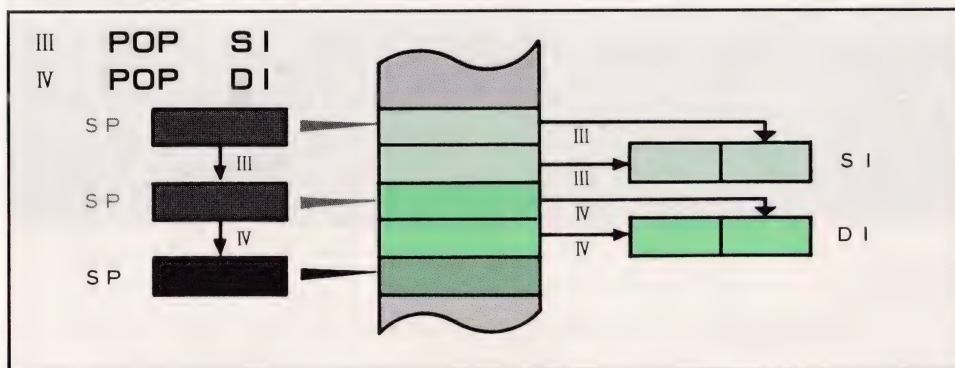
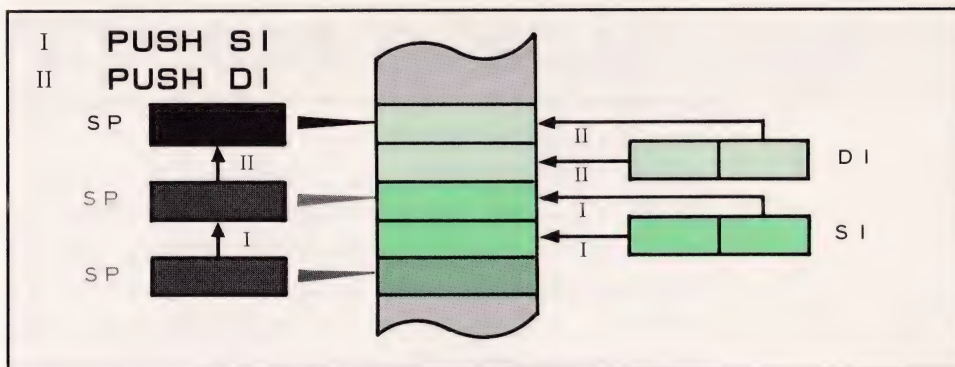
```
PUSH     SI
```

```
PUSH     DI
```

```
POP      SI
```

```
POP      DI
```

となります。



レジスタ退避を利用してレジスタのデータを交換するのは特殊な手法で、一般的にはXCHGを使います。先ほどの場合は、

XCHG

XCHG DI, SI

と表現することができます。

XCHGは、EXCHANGEの意味を持ち、2つのレジスタ間でデータを交換するための命令ですが、

XCHG AX, AX

には、少しかわった使い方があります。ために、モニタのアセンブラによって上記の命令をアセンブルしてみましょう。

hJc1800

hJa0000

0000 90

0001

xchg ax,ax

オブジェクト・コードは90H

アドレスのすぐ右側に表示された機械語のコード(オブジェクト・コードと呼ぶ)は90Hであることがわかります。

同様に、NOPという命令をアセンブルしてみます。NOPはNO OPERATIONの意味で、わずかな時間を浪費するのみで何の操作も行

NOP

なわないために使用します。なぜ、このように何もしない命令が必要なのか疑問が生じるかも知れませんが、プログラムによっては何もしない命令が要求される場合があります。特に本書のように本格的なアセンブラを使用しない場合には、デバッグによってプログラム全体の長さが変わった時に大変です。そこで長いプログラムをつくる場合には、プログラムの切れ目となる部分に事前にいくつかのNOPを入れておけばプログラムの途中で命令を挿入する時に便利ですし、逆にデバッグでメモリに余裕が出た場合には、NOPでメモリを埋めておくことができます。また、正確なタイミングが必要となるコントロール・プログラムでは時間を調整するためにNOPを使用することがあります。

```
h]c1800
h]a0000
0000 90
0001 90
0002
```

```
xchg ax,ax
nop
```

両者ともオブジェクト・コードが90H

上図でわかるように、NOPのオブジェクト・コードも90Hです。

80系の8ビットCPU(8080、8085、Z80など)では、NOPのために専用のコード(00H)が割り当てられていましたが、8086では、レジスタAXの内容をレジスタAX自身の内容と入れ換える命令で代用しなくてはなりません。

PC-9801を使っている皆さんのほとんどが、BASIC(高級言語)からアセンブリ言語への経路で、パーソナル・コンピュータの学習を進めて来られたことと思います。

BASICなど、手軽に扱える言語が標準装備されていることは、パーソナル・コンピュータの普及やビジネス面への応用を考えると非常に素晴らしいことなのですが、BASICの命令を憶えている点か、かえってアセンブリ言語への理解をおくらせている面があるかも知れません。

N88-BASIC(86)の命令と対比することのできるアセンブリ言語の命令も数多く存在しますが、特にスタックの考え方などはBASICには全くありませんから、N88-BASIC(86)に慣れてしまった頭を完全に切り換える必要が生じてきます。

「必要は発明の母」とは良く言ったもので、手軽に使えるコンパイラ型の高級言語、各メーカーから提供される数々のパッケージ・ソフトなどを見ていると、アセンブリ言語を学ぶ意欲もうすれてしまうのではないのでしょうか？

3.12 サブルーチン・コールとレジスタ退避

サブルーチンをコールする際に、メイン・ルーチンへのもどり先アドレスがスタックに退避されることは、3.8節で触れました。

8086では、レジスタ退避命令の実行時に使用されるスタックと、サブルーチン・コールの際にアドレスが退避されるスタックが同じであるため、サブルーチン内外でのレジスタ退避命令の使用に関して注意を要する場合があります。

次に示すサブルーチンは、レジスタAX、レジスタBXそしてレジスタCXの内容をレジスタに退避するサブルーチンです。

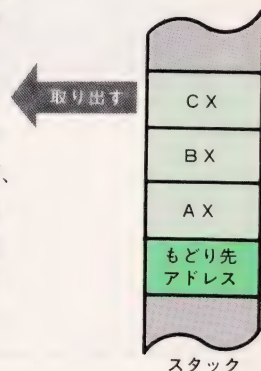
PUSH	AX
PUSH	BX
PUSH	CX
RET	

このサブルーチンは、一見何の問題もなく正常に動作するように思えます。しかし、最後のリターン命令 (RET) でメイン・ルーチンへもどることは決してありません。なぜだかわかりますか？

リターン命令の動作を思い出してください。CPUはサブルーチンからもどるために、スタックからもどり先のアドレスを取り出しますが、スタックから取り出されるのは直前に退避されたレジスタCXの内容です。したがって、メイン・ルーチンへもどるかわりに、レジスタCXに格納されていた16ビットのデータをアドレスとみなして、そのアドレスにジャンプしてしまうため、ほとんどの場合には暴走してしまいます。

メイン・ルーチンにもどしたくない場合などに、わざとスタックの内容を変更する手法もありますが、あまりすすめられる方法ではありません。

このような問題は、サブルーチン・コール用のスタックとレジスタ退避用のスタックを分離すればすむことで、実際に2種類のスタックを用意しているCPU (6809、68000など) も存在します。



3.13 8086の乗除算命令

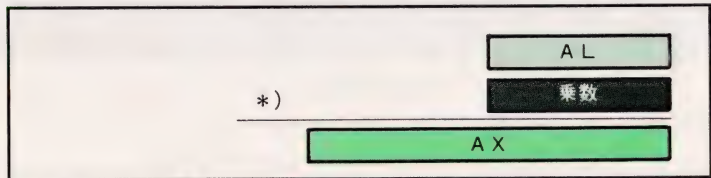
Z80 (8080) の頃と同レベルの命令ばかり使っているのは面白味が半減しますので、ここでは、8086特有の乗除算命令を使用してみましょう。

MUL
DIV
IMUL
IDIV

8086には乗除算命令として、MUL(無符号乗算)、DIV (無符号除算)、IMUL(有符号乗算)、IDIV (有符号除算)の4種が用意されています。そして、これらの命令では、他の演算命令と異なり、レジスタの使用目的(割り当て)がある程度限定されています。

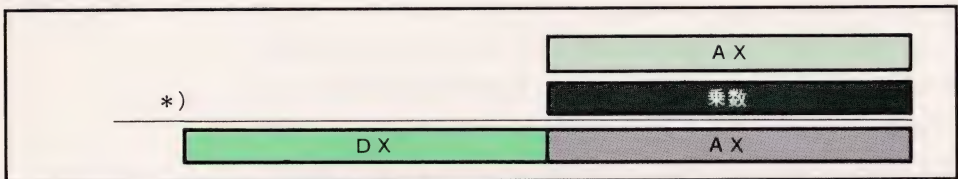
●8ビット乗算命令

8ビットの乗算命令では、被乗数をレジスタALに与えなければならず、積はレジスタAXに16ビット長で求められます。乗数は、任意の8ビット・レジスタまたはメモリを指定することができます。



●16ビット乗算命令

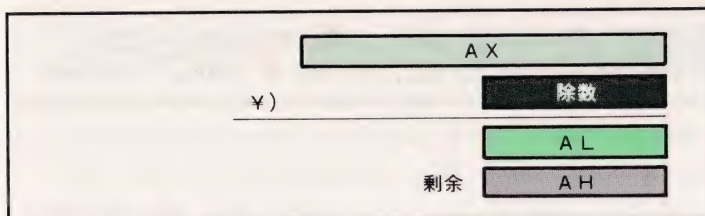
16ビットの乗算命令では、被乗数をレジスタAXに与えなければならず、積は上位16ビットがレジスタDXに、下位16ビットがレジスタAXに、計32ビット長で求められます。乗数は、任意の16ビット・レジスタまたは2バイトにわたるメモリ(逆ワード)を指定することができます。



●8ビット除算命令

8ビットの除算命令では、被除数をレジスタAXに与えなければならず、商はレジスタALに、剰余はレジスタAHに求められます。除数は、任意の8ビット・レジスタまたはメモリを指定することができます。

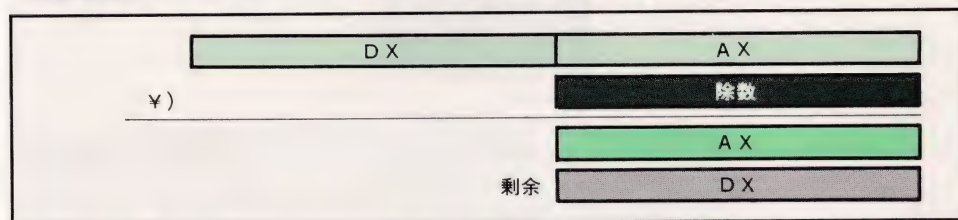
もし、除算結果がオーバフローした場合には、タイプ0の割り込みが発生し、割り込み処理ルーチンへ分岐します。



◎16ビット除算命令

16ビットの除算命令では、被除数の上位16ビットをレジスタDXに、下位16ビットをレジスタAXに与えなければならず、商はレジスタAXに、剰余はレジスタDXに求められます。除数は、任意の16ビット・レジスタまたは2バイトにわたるメモリ（逆ワード）を指定することができます。

除算結果のオーバーフローでタイプ0の割り込みが発生します。



ここで注意しなければならないのが、有符号整数データの除算命令 (IDIV) を実行した結果において、商と剰余の組み合わせが複数存在する可能性のあることです。

たとえば、被除数が10で、除数が-4である時、除算結果は次の2通りの組み合わせが考えられます。

- ① 商……-2 剰余……+2
- ② 商……-3 剰余……-2

このような場合、8086は被除数と剰余の符号が一致するように（上記例では①を選択）設計されています。

被除数	除数	商	剰余
+	+	+	+
+	-	-	+
-	+	-	-
-	-	+	-

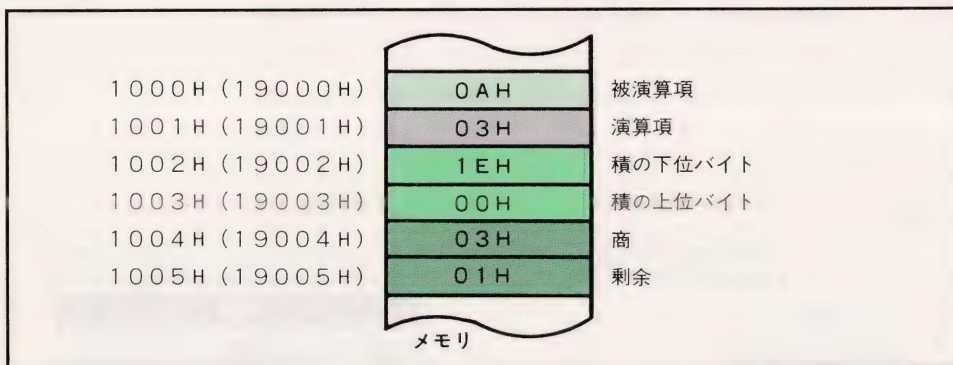
3・14 乗除算命令を使ったプログラム

無符号乗算命令

無符号除算命令

乗除算命令の中でも最も基本的な、8ビットの無符号乗算命令および8ビットの無符号除算命令を使ってみました。

データ・エリアはコード・セグメント中に、下図のように割り当ててあります。



メモリ・マップからもわかるように、8ビットのデータ同士で乗算および除算を行なう非常に簡単なプログラムです。さっそくコーディングしたプログラム・リストを示します。

0000 2E	CS:			
0001 A00010	MOV	AL,[1000]		被乗数をレジスタALに得る
0004 2E	CS:			
0005 F6260110	MUL	BYTE [1001]		1001H番地の乗数を掛ける
0009 2E	CS:			
000A A30210	MOV	[1002],AX		積を1002H～1003H番地に格納する
000D B400	MOV	AH,00		
000F 2E	CS:			
0010 A00010	MOV	AL,[1000]		被除数をレジスタAXに得る
0013 2E	CS:			
0014 F6360110	DIV	BYTE [1001]		1001H番地の除数で割る
0018 2E	CS:			
0019 A30410	MOV	[1004],AX		商を1004H番地に剰余を1005H番地に格納する
001C F4	HLT			ホールド

乗算命令 (MUL) と除算命令 (DIV) 以外に特別な手法は用いていませんが、乗除算命令のオペランドに含まれた、BYTEが気にかかります。

BYTE
WORD

このBYTEは、命令が8ビットのバイト・データを扱う命令であることを、アセンブラに対して指示するためにつけるもので、機械語モニ

タの内蔵アセンブラがオブジェクト・コードを生成する際に、8ビットの命令を生成するべきか、16ビットの命令を生成するべきかの判断を行なう基準となります。したがって今回のように、命令を記述したのみでは、それが8ビットのデータを扱うものなのか、16ビットのものなのかは確実に判断できない場合には、BYTEまたはWORDを付加して指示しなければ、アセンブラが受け付けてくれません。

それでは、どのような場合にアセンブラによる判断が困難なのでしょう。か？ 無符号乗算命令 (MUL) を例にとりて考えてみましょう。

前節でも述べたように、MULはオペランドで乗数の指定を行ない、乗数としては、下表のレジスタまたはメモリ上に格納されたデータが許されています。

8ビット乗数	AH	AL	BH	BL	CH	CL	DH	DL
16ビット乗数	AX	BX	CX	DX	SP	BP	SI	DI

これらのレジスタは8ビットと16ビットの区別が明確になされているため、オペランドとしてレジスタを指定した場合には、特別にバイト・データかワード・データかの指示を与える必要はありません。

8ビット乗算命令の例		16ビット乗算命令の例	
MUL	AH	MUL	AX
MUL	AL	MUL	BX
MUL	BL	MUL	CX
MUL	CL	MUL	SI
MUL	DH	MUL	DI

しかし、オペランドにメモリを指定する場合はどうなるのでしょうか？ 単に、

```
MUL      [1000H]
```

と記述した場合に、乗数が1000H番地に格納されたバイト・データであるのか、それとも1000H番地と1001H番地に逆ワードで渡るワード・データであるのかを区別することができません。

したがって、オペランドの対象としてメモリを指定する場合には、次のようにBYTEまたはWORDを付加することが必要です。

```
MUL      BYTE [1000H]
```

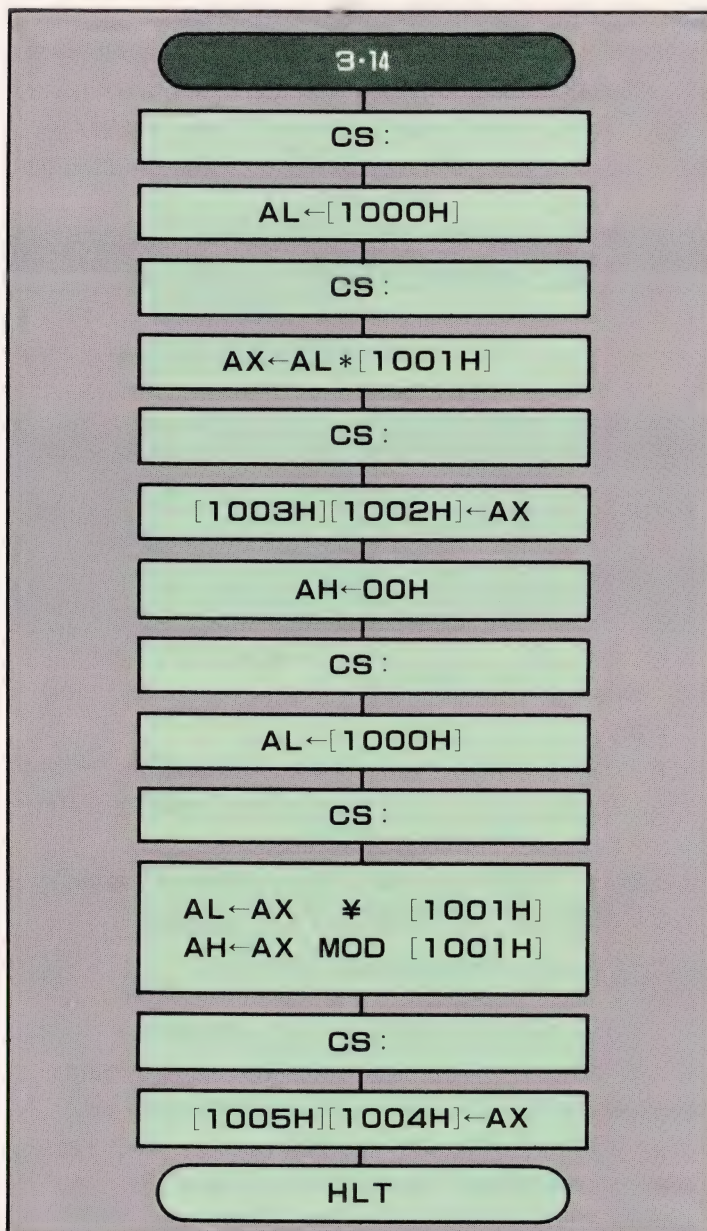
```
MUL      WORD [1000H]
```

ここではMULを例にして説明しましたが、単項演算のオペランドとしてメモリ上のデータが許されている等の場合には、BYTEまたはWORDの指示を要することがあります。多少手間がかかりますが、8086が全ての命令に対して8ビットの場合と16ビットの場合を両方ともサポートしている性質上、仕方ないことと言えましょう。

また、注意していただきたいのが、バイトとワードの指示方法がア

センブラによって異なることです。比較的簡単なアセンブラでは、N₈₈-BASIC(86)機械語モニタのアセンブラのように、BYTEとWORDを付加して区別する場合がありますが、統一化されているわけではありません。

話題をもとにもどして、フローチャートを示します。



¥……………整数除算
MOD…………整数剰余

h]c1800		扱うセグメントを設定する
h]a0000		
0000 2E	CS:	
0001 A00010	mov al,[1000]	
0004 2E	CS:	
0005 F6260110	mul byte [1001]	
0009 2E	CS:	
000A A30210	mov [1002],ax	
000D B400	mov ah,00	プログラムを入力する
000F 2E	CS:	
0010 A00010	mov al,[1000]	
0013 2E	CS:	
0014 F6360110	div byte [1001]	
0018 2E	CS:	
0019 A30410	mov [1004],ax	
001C F4	hlt	
001D		
h]s1000		データをセットする
1000 FF-0a 00-03 00-00 00-00 00-00 00-00		
h]g0000,001c		プログラムを実行する
*1800:001C		
h]d1000,1005		実行結果を確認する
1000 0A 03 1E 00 03 01		
h]■		剰余
		商
		積の上位バイト
		積の下位バイト
		演算項
		被演算項

h]s1000		エラーの起こるデータをセットする
1000 0A-0a 03-00 1E-00 00-00 03-00 01-00		
h]g0000,001c		プログラムを実行する
*1800:001C		
h]d1000,1005		実行結果を確認する
1000 0A 00 00 00 0A 00		
h]■		

上図はプログラムを実行した様子です。

上側の実行例では正常な実行結果が得られていますが、下側の場合には除数が00Hであるためオーバーフローを起こしてタイプ0の割り込みが発生します。しかし、N88-BASIC(86)の機械語モニタでは、タイプ0の割り込みは無視され、割り込み処理ルーチンからすぐにもどってくるので考慮する必要はありません。

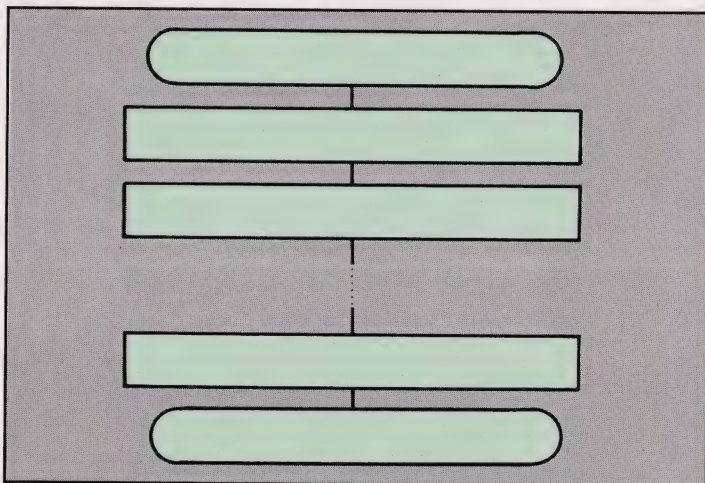
3・15 第3章のまとめ

第3章では、「簡単なプログラム」と題して、8086の命令を使ったプログラムの数々を紹介して来ました。命令の説明やサンプル・プログラムの実行方法なども、できる限り細かく述べたつもりですが、ご理解いただけたでしょうか？

単純なプログラムが多く、読みあきた皆さんもおられることでしょうが、何事も基本が大切です。しっかりと理解された上で次の章に進んでいただきたいと思います。

さて、本章のサンプル・プログラムには全てフローチャートを示しましたが、これらのフローチャートをながめてみると、規則性のあることに気がつきます。

そうです。本書のサンプル・プログラムは、全てが上から下へ流れるだけで、フローチャート中の箱も最初と最後を除いては長方形のみで構成されており、一般のフローチャートではよく見られるようなひし形の箱も使用していません。



このように、上から下へと流れるだけの構造を、逐次構造とか順次構造などと呼びます。したがって本章のタイトルにある「簡単なプログラム」を、少し気取って「逐次構造プログラム」などと呼んでも良いわけです。

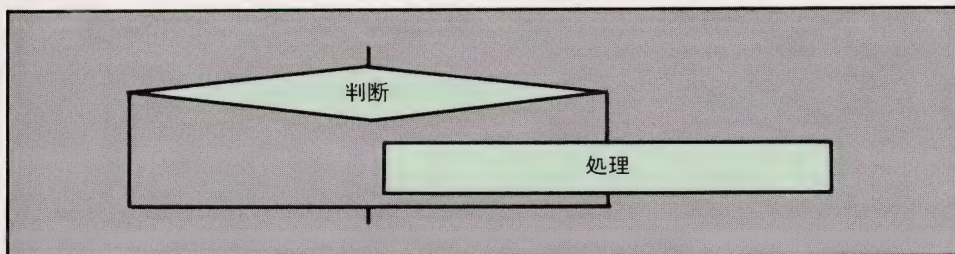
逐次構造
順次構造

4章 分岐のあるプログラム

4.0 分岐のあるプログラム

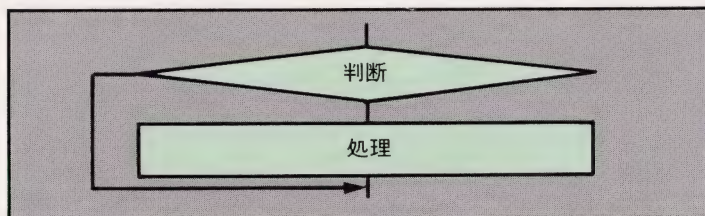
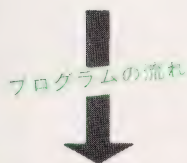
第3章では逐次構造のプログラムを題材にして話を進めて来ましたが、本章が題材として扱うのは、選択構造のプログラムです。

最も簡単な選択構造のフローチャートを示します。



このフローチャートでは、条件判断によって、次の処理を行なうかどうかを選択しており、左側に分岐した場合には処理を行わず、右側に分岐した場合には処理を行なう構造を持っています。

構造化の立場で考えれば、上記の書式でフローチャートを記述するのが当然ですが、この書式のフローチャートからアセンブリ言語のプログラムをコーディングするには多少無理があります。そこで上記のフローチャートをコーディングしやすいように書き直してみました。



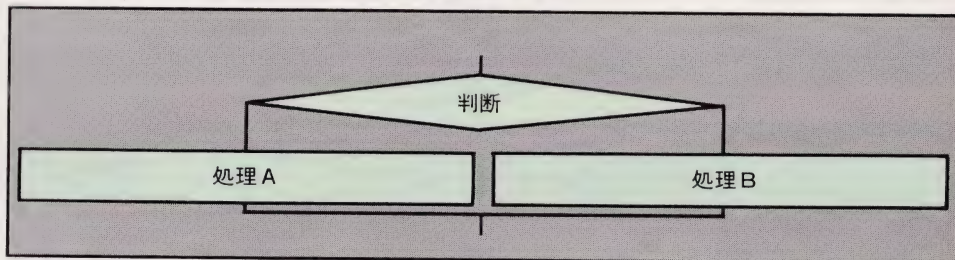
先のフローチャートと比較して、処理している内容自体は直感的にわかりにくいですが、アセンブリ言語のプログラム・リストと対比した場合に、アドレスの流れにそって対応させられることがわかると思います。

皆さんもはじめのうちは、上記のフローチャートを一度コーディングしやすい形に変形してからコーディングした方が、ミスの発生も防げて良いかも知れません。

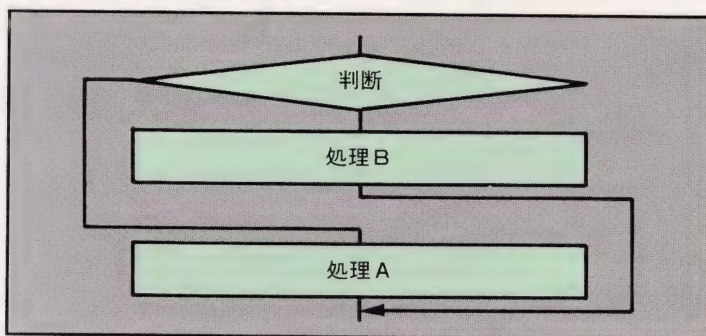
このようなアセンブリ言語の入門書という立場から記述したため、本書掲載のフローチャート全ては、構造化 (STRUCTURED PROGRAMMING) よりも、コーディングの行ないやすさおよびプログラム・リストとの対比性を重視しています。

あらかじめ、ご理解いただきたいと思います。

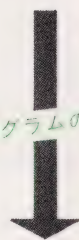
同様のことを、今度は一般的な条件分岐の場合について考えてみましょう。ただし、ここで言う一般的な条件分岐とは、条件判断で分岐した左右どちら側にも何らかの処理を伴うものです。



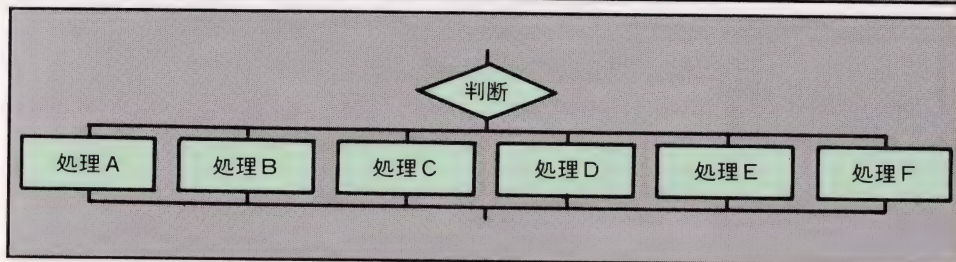
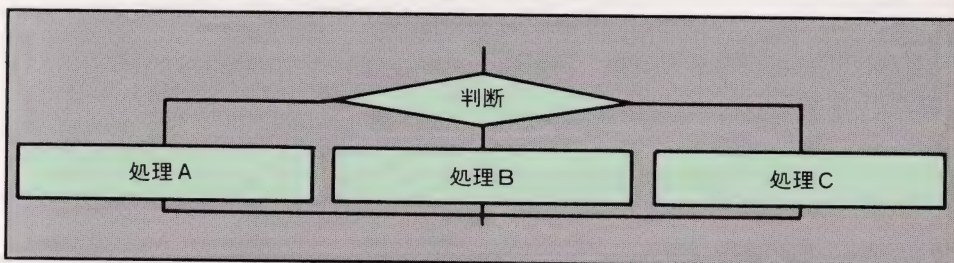
このフローチャートをアセンブリ言語でコーディングしやすいように変形したものを示します。

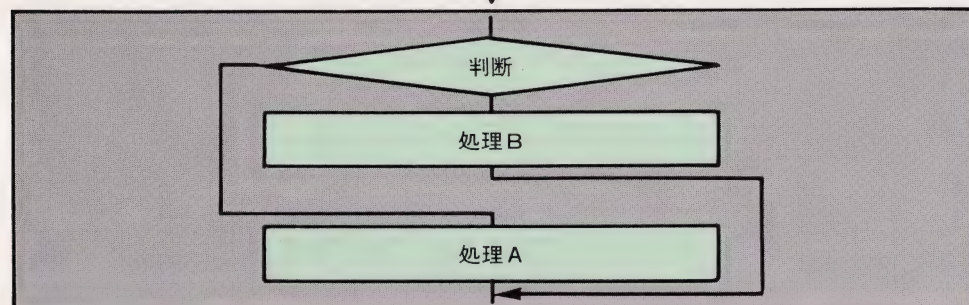
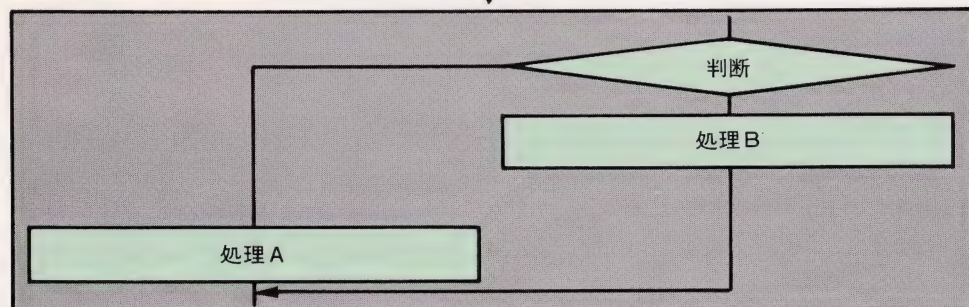
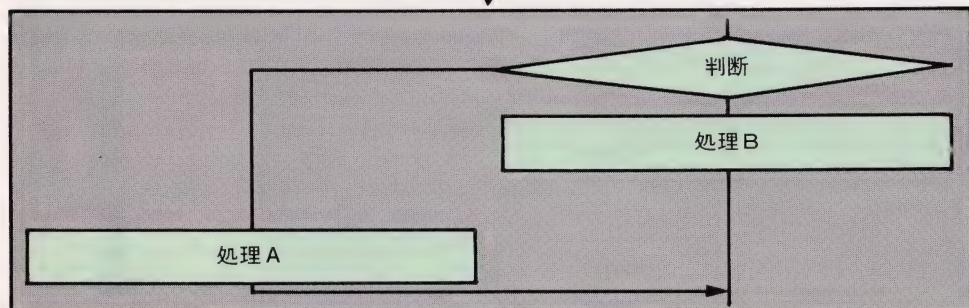
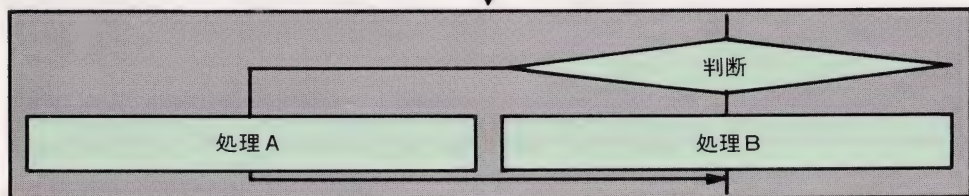
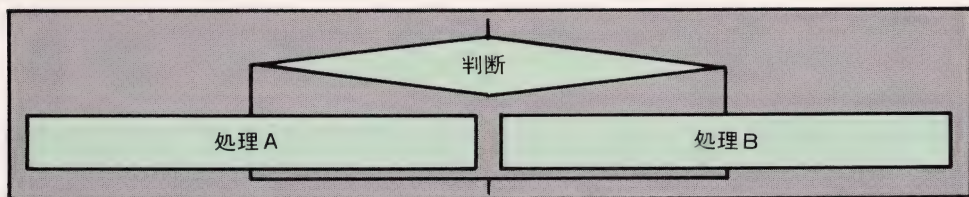


プログラムの流れ



また、フローチャートを変形して行く過程を次ページに示します。さらに、3通り以上に分岐する例として、3通りおよび6通りに分岐するフローチャートをあげておきますのでご参照ください。





4.1 ゼロ・フラグ(ZF)による分岐

さっそく分岐のあるプログラムをつくってみました。

0000 2E	CS:		
0001 A00010	MOV	AL,[1000]	1000H番地のデータを得る
0004 3C0A	CMP	AL,0A	
0006 7504	JNE	000C	データが0AHでなければ000CH番地へジャンプする
0008 2E	CS:		
0009 A20110	MOV	[1001],AL	データが0AHであれば1001H番地へコピーする
000C F4	HLT		ホールド

このプログラムは、コード・セグメントの1000H番地の内容が0AH (10進数では10) であれば、1001H番地にコピーするプログラムです。

データが0AHであるかどうかを調べるために、

CMP AL, 0AH

CMP

という命令を使っていますが、これはCOMPARE (比較する) の意味を持ち、レジスタALの内容から0AHを減じた結果をフラグ(F)にセットします。ただし、変化するのはフラグのみで、減算結果自体を求めることはできません。

ここでは、次の命令、

JNE 000CH

JNE

を実行し、ゼロ・フラグ (ZF) を調べることによって、データが0AHと等しくなければ000CH番地にジャンプします。すなわち、データが0AHと等しい場合にのみ、1001H番地へのコピー命令

CS:

MOV [1000H], AL

を実行することになります。

このJNEは、JUMP IF NOT EQUAL (等しくなければジャンプする) の意で、次にあげる命令JUMP IF NOT ZEROも全く同じものです。

JNZ 000CH

JNZ

8086には、このJNEとJNZのように全く同じオブジェクト・コード (機械語コード) に対して複数のニーモニックが用意されている場合があります、非常に便利です。

このように、フラグ(F)の状態によってジャンプを行なう命令を条件分岐命令 (条件ジャンプ命令) と呼びますが、次ページに8086に用意された16種の条件分岐命令を示します。

条件分岐命令

表中で、上 (ABOVE) 下 (BELOW) と大 (GREATER) 小 (LESS) の相違は、前者の上下が無符号データの比較を対象とするのに対して、後者の大小が有符号データを対象としている点です。すなわち、バイト・データ 01H と FFH を比較した場合には 01H は FFH より下であり、かつ、大であることになりましたが、これは FFH を有符号データとみなした場合には 10 進数の -1 と等価なためです。

命令	名称	意味
JE	JUMP IF EQUAL	等しければ分岐
JNE	JUMP IF NOT EQUAL	等しくなければ分岐
JB	JUMP IF BELOW	下ならば分岐
JNB	JUMP IF NOT BELOW	下でなければ分岐
JA	JUMP IF ABOVE	上ならば分岐
JNA	JUMP IF NOT ABOVE	上でなければ分岐
JL	JUMP IF LESS	小ならば分岐
JNL	JUMP IF NOT LESS	小でなければ分岐
JG	JUMP IF GREATER	大ならば分岐
JNG	JUMP IF NOT GREATER	大でなければ分岐
JS	JUMP IF SIGN	負ならば分岐
JNS	JUMP IF NOT SIGN	負でなければ分岐
JO	JUMP IF OVERFLOW	オーバーフローならば分岐
JNO	JUMP IF NOT OVERFLOW	オーバーフローでなければ分岐
JP	JUMP IF PARITY	パリティならば分岐
JNP	JUMP IF NOT PARITY	パリティでなければ分岐

上記命令のいくつかは、前述した様に別の記述が許されています。

命令	名称
JE	JZ JUMP IF ZERO
JNE	JNZ JUMP IF NOT ZERO
JB	JNAE JUMP IF NOT ABOVE NOR EQUAL
JNB	JAE JUMP IF ABOVE OR EQUAL
JA	JNBE JUMP IF NOT BELOW NOR EQUAL
JNA	JBE JUMP IF BELOW OR EQUAL
JL	JNGE JUMP IF NOT GREATER NOR EQUAL
JNL	JGE JUMP IF GREATER OR EQUAL
JG	JNLE JUMP IF NOT LESS NOR EQUAL
JNG	JLE JUMP IF LESS OR EQUAL
JP	JPE JUMP IF PARITY EVEN
JNP	JPO JUMP IF PARITY ODD

これらの条件分岐命令は、あくまでもフラグ(F)の状態のみで条件判断を行なうことに注意してください。つまり、

```
CMP      AH, AL
JE       0000H
```

は、レジスタAHの内容とレジスタALの内容が等しかった場合にのみ0000H番地に分岐（ジャンプ）しますが、分岐命令(JE)はレジスタAHおよびレジスタALの内容を一切感知しません。直前の比較命令(CMP)によって変化したゼロ・フラグ(ZF)の状態のみが条件判断の対象となり、それが1にセットされている場合には0000H番地に分岐します。

したがって、比較命令(CMP)と条件分岐命令(上記の例ではJE)の間に、判断に関係のない命令が挿入されていても、それらの命令がゼロ・フラグ(ZF)に影響を与えさえしなければ、正常に条件分岐命令が実行されます。

```
CMP      AH, AL
```

関係のない命令。ただしゼロ・フラグ(ZF)に影響を与えてはならない

```
JE       0000H
```

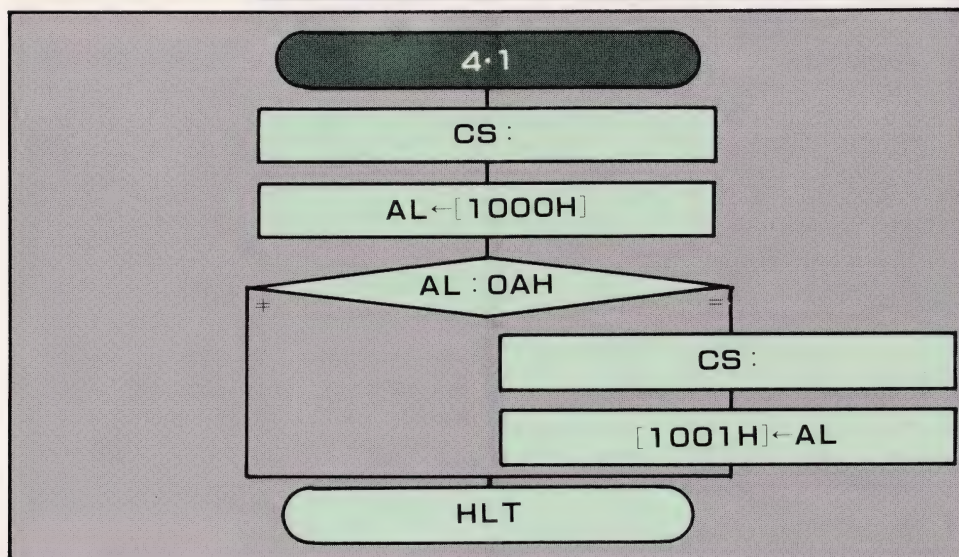
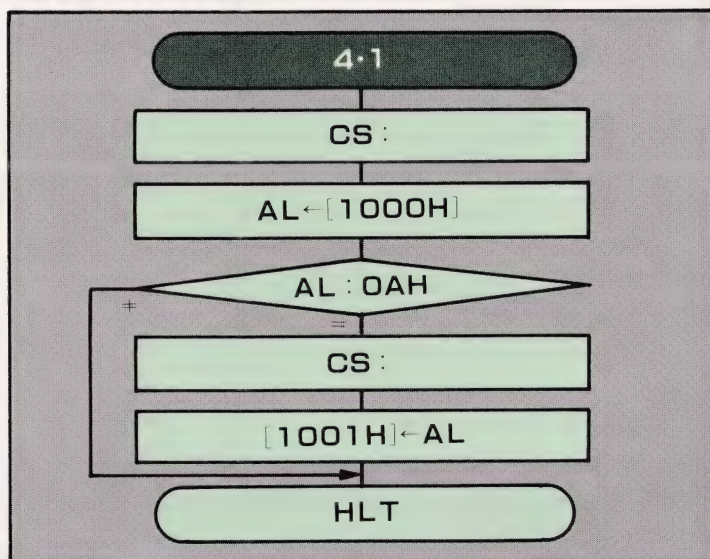
次表に、それぞれの条件分岐命令において、条件が成立し分岐が行なわれる時点でのフラグの状態を示します。

命令		フラグの状態
JE	JZ	ZF=1
JNE	JNZ	ZF=0
JB	JNAE	CF=1
JNB	JAE	CF=0
JA	JNBE	$(CF \vee ZF) = 0$
JNA	JBE	$(CF \vee ZF) = 1$
JL	JNGE	$(SF \vee OF) = 1$
JNL	JGE	$(SF \vee OF) = 0$
JG	JNLE	$((SF \vee OF) \vee ZF) = 0$
JNG	JLE	$((SF \vee OF) \vee ZF) = 1$
JS		SF=1
JNS		SF=0
JO		OF=1
JNO		OF=0
JP	JPE	PF=1
JNP	JPO	PF=0

68系のCPUには全ての命令が全てのフラグに対して影響を与えるものもありますが、8086も含めた一般的なCPUでは特定の命令が特定のフラグにのみ影響を与えます。どの命令を実行することによって、どのフラグ(F)が影響を受けるのかは、本書の巻末付録をご参照ください。

前置きが長くなりましたが、フローチャートを示します。

下側が一般的なフローチャート、上側が今後本書で扱って行く形式に書き直したものです。



プログラムを実行してみましょう。

1000H番地のデータが0AHである場合にのみ、1001H番地に0AHをコピーするプログラムですから、1000H番地に0AHを含めたいろいろな値を入れて、実行結果を確認しなければなりません。

私の場合には、0AHより小さいデータ(05H)と0AH、それには0AHより大きいデータ(11H)の3通りをためしてみました。

下図の実行結果を御覧になれば、データに0AHを与えた場合にのみ、1001H番地にデータがコピーされている様子が良くわかると思います。

```
h)c1800
h)a0000
0000 2E          cs:
0001 A00010      mov al,[1000]
0004 3C0A        cmp al,0a
0006 7504        jne 000c
0008 2E          cs:
0009 A20110      mov [1001],al
000C F4          hlt
000D
h)s1000
1000 00-05 00-00
h)g0000,000c
*1800:000C
h)d1000,1001
1000 05 00
h)s1000
1000 05-0a 00-00
h)g0000,000c
*1800:000C
h)d1000,1001
1000 0A 0A
h)s1000
1000 0A-11 0A-00
h)g0000,000c
*1800:000C
h)d1000,1001
1000 11 00
h)■
```

プログラム・リストを入力する

0

データが0AHより小さい場合

データが0AHの場合(コピーされている)

データが0AHより大きい場合

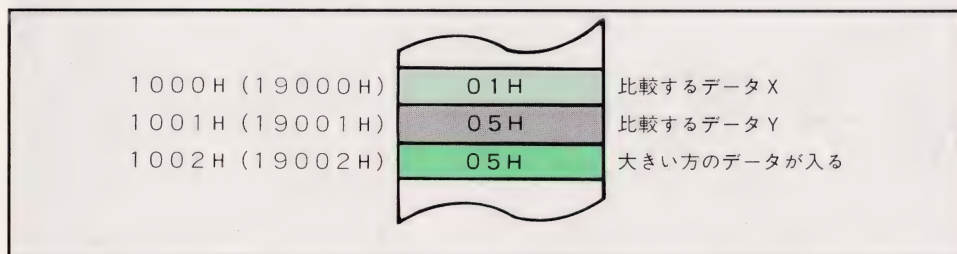
4.2 キャリ・フラグ(CF)による分岐

2つのデータが等しいかどうかを調べる方法は前節で説明しました。

それでは、2つの数値データがある場合に、どちらが大きいのかを調べるにはどうしたら良いのでしょうか？ 前の節ではゼロ・フラグ(ZF)を利用した条件判断を行ないましたが、本節ではキャリ・フラグ(CF)による条件分岐を考えてみましょう。

プログラムは、メモリ上に置かれた2つのバイト・データを無符号の数値として比較し、大きい(上の)方のデータを選び出すものです。

データ・エリアとしては、1000H番地からの3バイトを割り当てました。



プログラム・リストを示します。

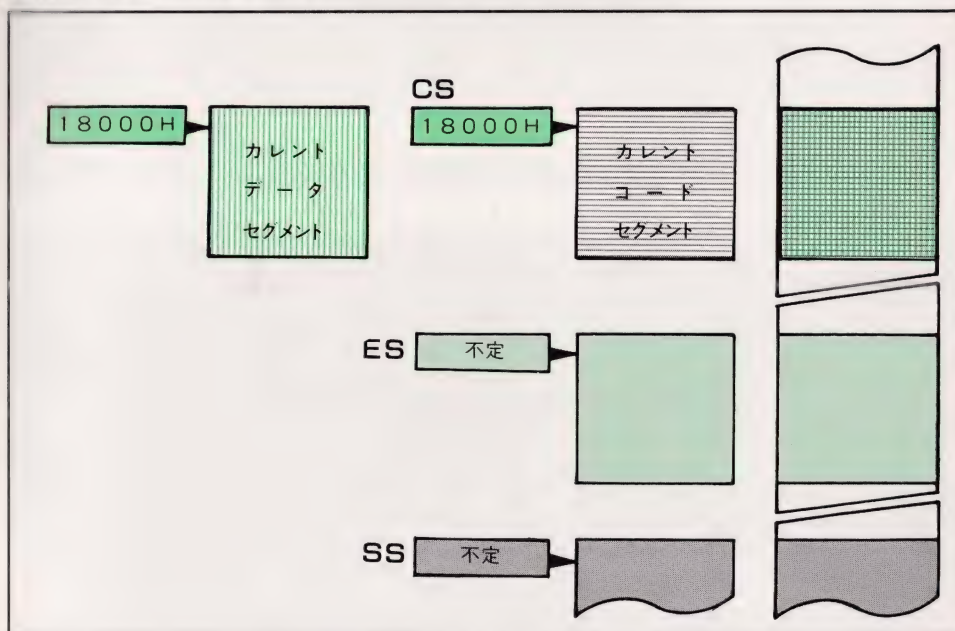
0000 8CC8	MOV	AX,CS	← カレント・コード・セグメントにデータ・セグメントを重ねる
0002 8ED8	MOV	DS,AX	
0004 A10010	MOV	AX,[1000]	← 2つのデータをレジスタAXおよびAHに得る
0007 38E0	CMP	AL,AH	← 2つのデータを比較する
0009 7302	JAE	0000	← レジスタALのデータが上ならば0000H番地へ分岐する
000B 88E0	MOV	AL,AH	← 下ならばレジスタAHのデータを移す
000D A20210	MOV	[1002],AL	← レジスタALの(大きい方の)データを与える
0010 F4	HLT		← ホールト

今まで本書で扱ってきたプログラムはコード・セグメント中に置かれたデータ・エリア(19000H番地以降)をアクセスする場合には全てCS:を付けていました。しかし、上記のプログラムではCS:を付けずに直接データ・エリアをアクセスしています。このような場合に8086はカレント・コード・セグメントではなく、カレント・データ・セグメントをアクセスしてしまい具合が良くありません。

そこで、最初の2つの命令で、あらかじめデータ・セグメントの位置するアドレスをコード・セグメントと全く同じアドレスに設定しておきますが、この設定は、コード・セグメント・レジスタ(CS)の内容をデータ・セグメント・レジスタ(DS)にコピーすることによって実現でき

ます。プログラム・リストでは一度レジスタAXを介してセグメント・アドレスをコピーしていますが、これはセグメント・レジスタ同士で直接にデータを転送する命令が用意されていないためです。

次図に、カレント・コード・セグメントとカレント・データ・セグメントが完全に重なった状態を示します。



データの比較は、1000H番地のデータをレジスタALに、1001H番地のデータをレジスタAHに、それぞれ得た後に、

CMPL AL, AH

を用いて行なっています。この時にレジスタAHのデータ（減数）の方が上であればキャリ・フラグ（CF）が1にセットされます。

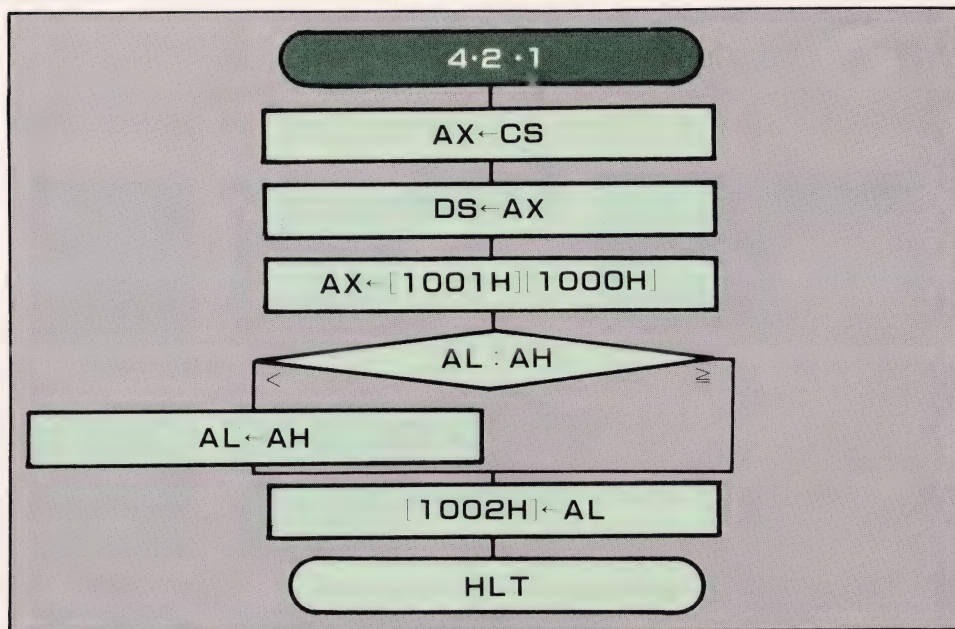
次に実行される

JAE 000DH

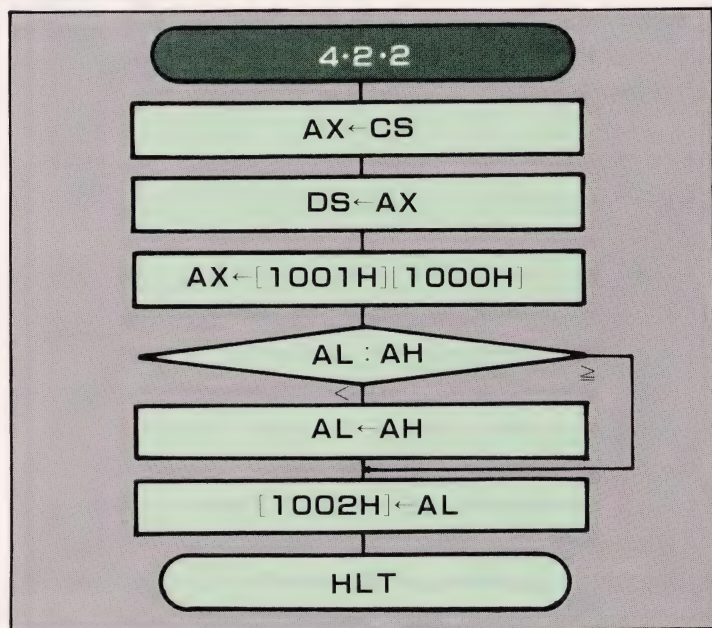
は、JUMP IF ABOVE OR EQUAL（上か等しければ分岐）の意味を持ちますが、具体的にはキャリ・フラグ（CF）の状態を判断するための命令で、それが0にリセットされている場合にのみ000DH番地への分岐を行いません。

ほかにも8086には多くのフラグが存在し、また各フラグの状態を判定する条件分岐命令も数多く用意されていますが、一般的に使用するのは、前節のゼロ・フラグ（ZF）と本節のキャリ・フラグ（CF）がほとんどです。他のフラグは後々学習して行くことにして、この2つのフラグだけでもしっかりと記憶しておきましょう！

フローチャートを示しますので、流れを追ってみてください。プログラム・リストだけではわかりにくい条件分岐のアルゴリズムも、フローチャートは非常に追いやすいと思います。



下図は、アセンブリ言語用に変形したフローチャートです。



1000H番地のデータをX、1001H番地のデータをYとした場合に、データXおよびデータYに関して3通りの組み合わせを考えて実行してみましょう。

X	Y	XYの関係	CF	分岐	結果
01H	05H	01H<05H	1	NO	05H
15H	00H	15H>00H	0	YES	15H
11H	11H	11H=11H	0	YES	11H

hjc1800

hja0000

0000 8CC8

0002 8ED8

0004 A10010

0007 38E0

0009 7302

000B 88E0

000D A20210

0010 F4

0011

hjs1000

1000 00-01 00-05 00-00

hjs0000,0010

*1800:0010

hjd1000,1002

1000 01 05 05

hjs1000

1000 01-15 05-00 05-ff

hjs0000,0010

*1800:0010

hjd1000,1002

1000 15 00 15

hjs1000

1000 15-11 00-11 15-00

hjs0000,0010

*1800:0010

hjd1000,1002

1000 11 11 11

hjm

mov ax,cs

mov ds,ax

mov ax,[1000]

cmp al,ah

jae 000d

mov al,ah

mov [1002],al

hlt

プログラムを入力する

データXが01HでデータYが05Hの場合

データXが15HでデータYが00Hの場合

データXデータY共に11Hの場合

4.3 分岐の組み合わせ

キャラクタ・コード表を眺めていると、PC-9801の持っているキャラクタを3種類に分類できることに気がつきます。

第1は、コントロール・コード(00H~1FH)で、これらのキャラクタは、単に表示または印字する以外の機能を持っていますから、他のキャラクタと区別して当然と言えます。

日本のパーソナル・コンピュータ界では、80H~FFHのキャラクタをカナ文字やグラフィック文字に割り当てるのが一般的になっていますが、本来アスキー(ASCII)コードと言えば00H~7FHのみを指し、80Hから後のキャラクタは使われないのが普通でした。そこで、残りのキャラクタ(20H~FFH)を、アルファベット・コード(20H~7FH)およびカナ・コード(80H~FFH)に分類してみました。

以上のような立場から本節で扱うプログラムは、1000H番地のバイト・データをキャラクタ・コードと見なして上記の3種類に分類し、1001H番地に分類番号を格納するプログラムです。

分類番号は、コントロール・コードの場合00H、アルファベット・

コントロール・キャラクタ

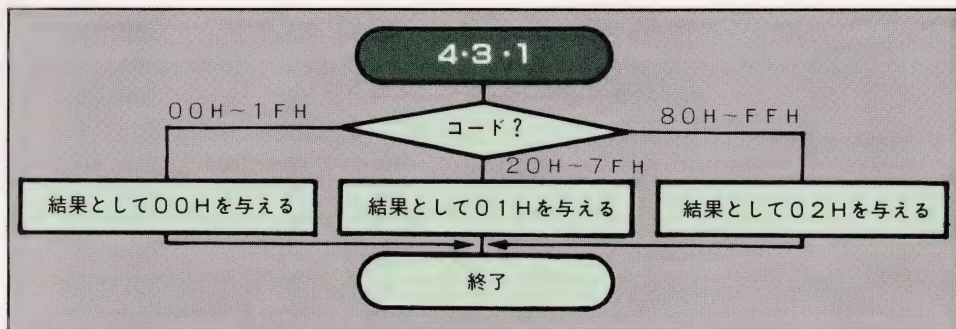
アルファベット・キャラクタ

カナ・キャラクタ

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		D _E		0	@	P		p				一	タ	ミ		×
1	S _H	D _I	!	I	A	Q	a	q				。	ア	チ	ム	円
2	S _X	D ₂	"	2	B	R	b	r				「	イ	ツ	メ	年
3	E _X	D ₃	#	3	C	S	c	s				」	ウ	テ	モ	月
4	E _T	D ₄	\$	4	D	T	d	t				,	エ	ト	ヤ	日
5	E _Q	N _K	%	5	E	U	e	u				・	オ	ナ	ユ	時
6	A _K	S _N	&	6	F	V	f	v				ヲ	カ	ニ	ヨ	分
7	B _L	E _B	'	7	G	W	g	w				ア	キ	ヌ	ラ	秒
8	B _S	C _N	(	8	H	X	h	x				「	イ	ク	ネ	リ
9	H _T	E _M	)	9	I	Y	i	y				ウ	ケ	ノ	ル	♥
A	L _F	S _B	*	:	J	Z	j	z				エ	コ	ハ	レ	♦
B	H _M	E _C	+	;	K	[	k	}				オ	サ	ヒ	ロ	♣
C	C _L	→	,	<	L	¥	!	!				ヤ	シ	フ	ワ	●
D	C _R	←	-	=	M	]	m	}				ユ	ス	ヘ	ン	○
E	S ₀	↑	.	>	N	^	n	~				ヨ	セ	ホ	.	◀
F	S ₁	↓	/	?	O	_	o					ツ	ソ	マ	°	▶

コードの場合01H、カナ・コードの場合02Hに、それぞれ決めておきます。

プログラムが少し複雑になりそうですから、ゼネラル・フローチャートを記述してみましょう。



プログラミングに際して、一般には、システム・フローチャート、ゼネラル・フローチャート、ディテール・フローチャートを記述します。システム・フローチャートはプログラムを実行することによって行なわせる作業を設定する段階（システム設計と呼ぶ）で、作業の流れを記すもので、直接プログラミングとは関係なく、いわばプログラムの仕様書に当たるものです。

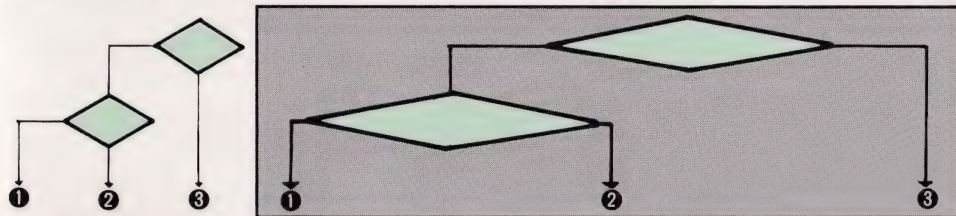
システム・フローチャート
↓
ゼネラル・フローチャート
↓
ディテール・フローチャート

実際にプログラムをつくる段階で作成するのは、ゼネラル・フローチャートおよびディテール・フローチャートで、前者のゼネラル・フローチャートはプログラムの流れを大まかにまとめた比較的簡潔なものであり、後者のディテール・フローチャートは実際に使用するプログラミング言語（本書ではアセンブリ言語）の命令のレベルまで落として詳細に記述したものです。

したがって、本書中に掲載した大部分のフローチャートはディテール・フローチャートとすることができます。

それでは、上記のゼネラル・フローチャートをディテール・フローチャートに落としてみましょう。

このプログラムをアセンブリ言語でつくる場合に問題になりそうなのが、3通りに分岐する部分ですが、条件分岐を下図のように組み合わせれば何とかなりそうです。



本ページにディテール・フローチャートを示し、さらにプログラミングを行ないやすいように変形したものを次ページに示します。2種類のフローチャートは見開きに対応させてありますが、両者を比較してみてください。

INC(インクリメント)

DEC(デクリメント)

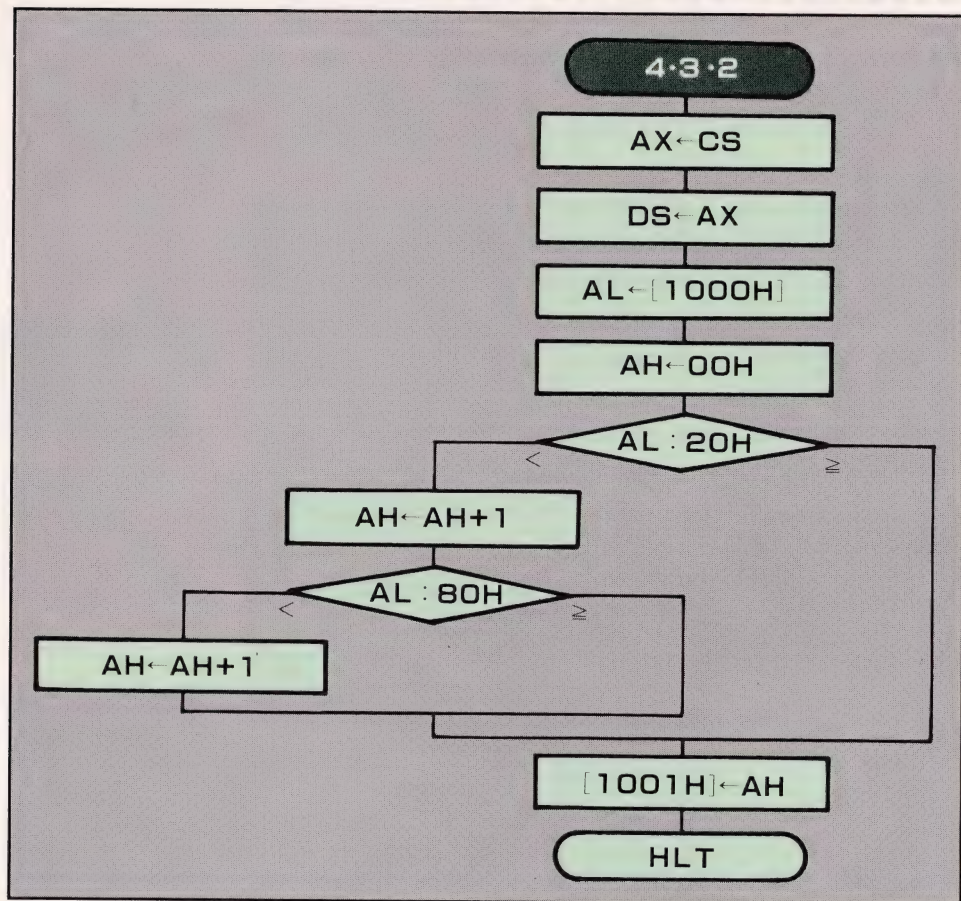
私も、このフローチャートをもとにしてコーディングを行ないましたが、レジスタAHの内容に1を加えるために、インクリメント命令(INC) を使いました。ここでは、

INC AH

で、レジスタAHの内容をインクリメント (増加) していますが、他のレジスタやメモリ上のデータをインクリメントすることもできます。

また、逆に1を減じるためのデクリメント命令 (DEC) も用意されています。

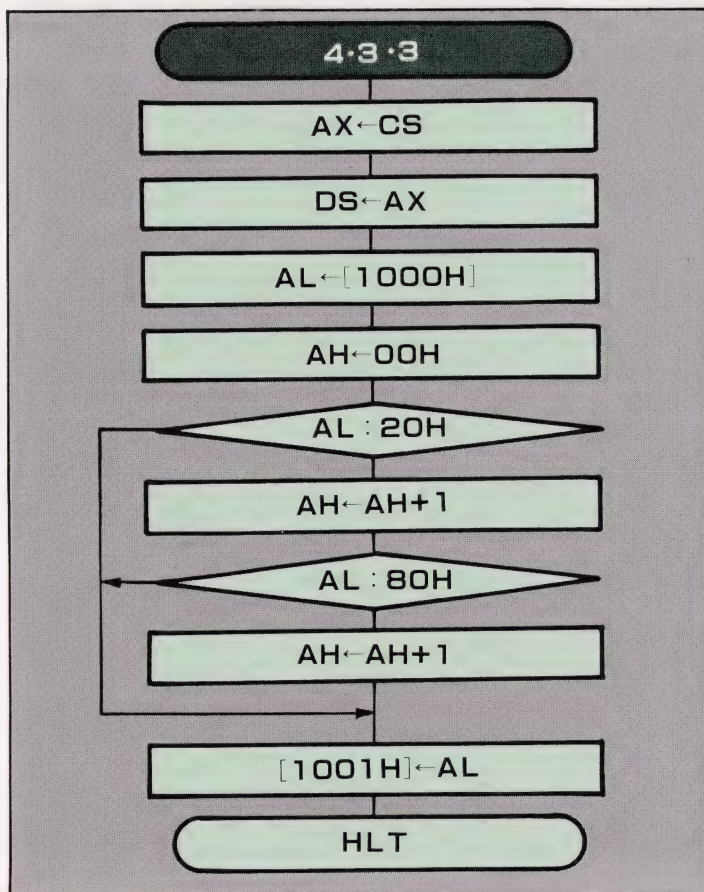
このプログラムでも比較命令 (CMP) と条件分岐命令を組み合わせ、2か所の分岐を行なっていますが、ここで使用した条件分岐命令は、



キャリ・フラグ (CF) が1の場合にのみ分岐を行なう命令 (JB) です。

JBは、JUMP IF BELOW (下ならば分岐) の意味で、この場合は、レジスタALのデータが第2オペランド (20Hまたは80H) より

0000 8CC8	MOV	AX,CS	カレント・コード・セグメントに データ・セグメントを重ねる
0002 8ED8	MOV	DS,AX	
0004 A00010	MOV	AL,[1000]	データをレジスタALに移す
0007 B400	MOV	AH,00	レジスタAHをクリアする
0009 3C20	CMP	AL,20	データが20Hより下であれば 0015H番地へ分岐する
000B 7208	JB	0015	
000D FEC4	INC	AH	レジスタAHの内容を01Hにする
000F 3C80	CMP	AL,80	データが80Hより下であれば 0015H番地へ分岐する
0011 7202	JB	0015	
0013 FEC4	INC	AH	レジスタAHの内容を02Hにする
0015 88260110	MOV	[1001],AH	結果(分岐番号)を与える
0019 F4	HLT		ホールド



も下の場合にのみ0015H番地に分岐します。

実行例を示します。今回は4通りの場合について実行してみました。

h]c1800		
h]a0000		
0000 8CC8	mov ax,cs	
0002 8ED8	mov ds,ax	
0004 A00010	mov al,[1000]	
0007 B400	mov ah,00	
0009 3C20	cmp al,20	
000B 7208	jb 0015	
000D FEC4	inc ah	プログラムを入力する
000F 3C80	cmp al,80	
0011 7202	jb 0015	
0013 FEC4	inc ah	
0015 88260110	mov [1001],ah	
0019 F4	hlt	
001A		
h]s1000		
1000 00-00 00-00		
h]g0000,0019		
*1800:0019		データが00Hの場合
h]d1000,1001		
1000 00 00		
h]s1000		
1000 20-2a 01-00		
h]g0000,0019		
*1800:0019		データが2AHの場合
h]d1000,1001		
1000 2A 01	*	
h]s1000		
1000 2A-7f 01-00		
h]g0000,0019		
*1800:0019		データが7FHの場合
h]d1000,1001		
1000 7F 01		
h]s1000		
1000 7F-ff 01-00		
h]g0000,0019		
*1800:0019		データがFEHの場合
h]d1000,1001		
1000 FF 02		
h]■		

5章 | くり返しのあるプログラム



5.0 くり返し処理のあるプログラム

逐次構造

選択構造

反復構造

基本三構造

構造化プログラミング

構造化言語

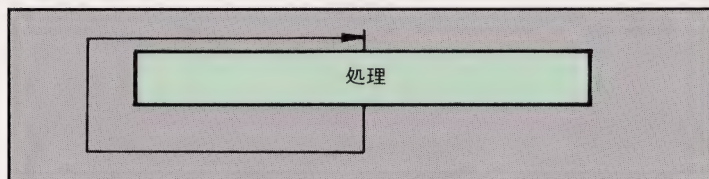
第3章では逐次構造（シーケンス）のプログラムを、第4章では選択構造（ブランチ）のプログラムを扱ってきましたが、本章では新しく反復構造（ループ）のプログラムを扱って行きたいと思います。

これら3つの構造を称して、基本三構造と呼び、全てのアルゴリズムはこの基本三構造のみで記述する事ができると言われています。この考え方が、最近一般的になりつつある構造化プログラミングの概念とも言えるもので、基本三構造をいかに自然に記述できるかを追求したプログラミング言語が、構造化言語です。

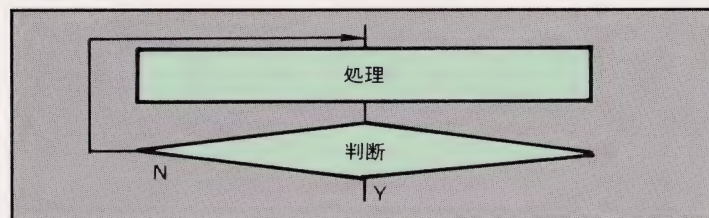
アセンブリ言語で構造的なプログラムをつくるのは、たいへんむずかしいことですが、これら3つの構造を学ぶことによって、基本的には全てのアルゴリズムをプログラミングし得ることがわかります。

ところで、反復構造とはどのような構造なのでしょう？

一定範囲内の作業をくり返して行なうためには、プログラムの流れを輪のようにすることが必要となりますが、これをループ（LOOP）と言います。

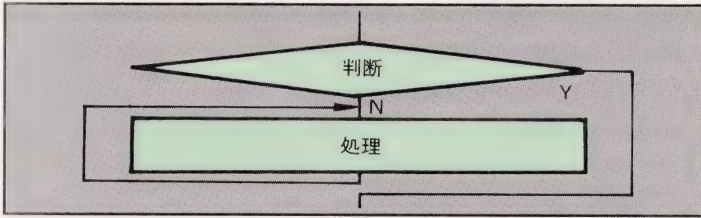


しかし、永久的に同じ作業をくり返したいことは、あまりありませんから、何らかの条件を満たした場合には、輪の中からぬけ出す必要があります。



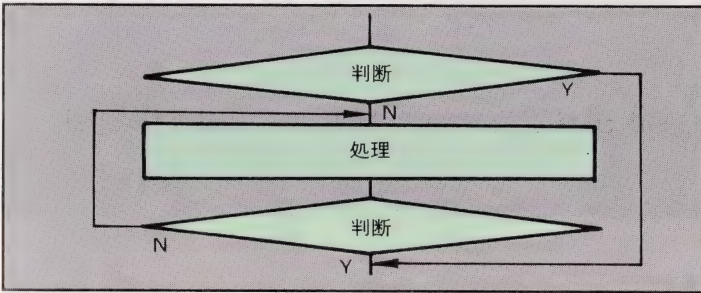
上記フローチャートのように、ループの最後で条件が満たされているかどうかを判断するのが、アセンブリ言語によるコーディングの場合の最も基本的な反復構造の例です。しかし、このフローチャートには問題点が1つ存在します。それは、はじめから条件が満たされており、1度も処理を行なう必要がない場合でも、最低1回は処理を行ってしまうことです。

このような問題点を克服するためには、一般的にループの前で条件判断を行なうことが知られています。



しかし、アセンブリ言語では、用意されている命令の都合から、ループの最後で条件判断を行なう場合がほとんどです。

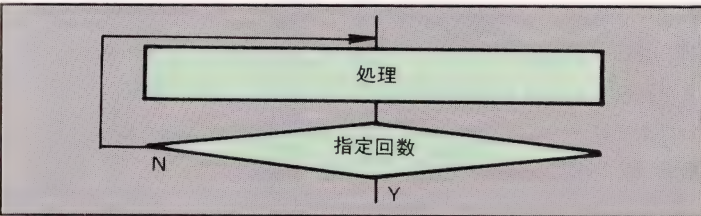
この場合には、前述の問題点に対するもう1つの対策手段として、2度手間にはなりますが、ループの最初と最後に条件判断を行なうことも考えられます。



アセンブリ言語によるプログラミングと高級言語(特に構造化言語)によるものとの間には、おのずから相違点が生じてくるものです。この辺がアセンブリ言語の手間のかかる部分であり、言い換えれば最もおもしろい所です。

こう書くと、アセンブリ言語によって行なえる範囲に制限があるように思えるかも知れませんが、高級言語で実現できることでアセンブリ言語で不可能なことは有り得ません。これは、高級言語そのものでさえも、機械語のコードによって構成されていることを考えれば明らかです。

なお、反復構造の条件判断では、処理の回数が一定回数に達したかどうかかを調べるのが最も一般的です。



5.1 基本的なループ処理

最初は、基本的な反復構造のプログラムとして、1000H番地から15バイトのメモリに55Hを書き込む方法を考えてみましょう。

モニタのFコマンド 1000H番地から100EH番地までの15バイトを55Hで埋めるのですから、機械語モニタのFコマンドによって、

F 1 0 0 0 . 1 0 0 E . 5 5 

を打ち込むのと同じ動作を、アセンブリ言語のプログラムで実現できれば良いわけです。

```
hJc1800
hJd1000,1010
1000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1010 00
hJf1000,100e,55
hJd1000,1010
1000 55 55 55 55 55 55 55 55 55 55 55 55 55 55 00      UUUUUUUUUUUUUUU
1010 00
hJ■
```

これを反復構造を使わないでコーディングしたものが、次ページのプログラム・リストです。さらにその次のページにフローチャートを示します。実行方法はすでにおわかりのことと思いますので、あえて示しませんが、興味のある方は、ご自分で走らせてみてください。

アドレッシング

このプログラムのおもしろい所は、レジスタBXによってアドレス指定（アドレッシング）を行なっていることで、今まではオフセット・アドレスを直接指定して、

MOV [1000H], 55H

などを行っていた部分を、このプログラムでは、

MOV BX, 1000H

MOV [BX], 55H

の2段階で、1000H番地に55Hを書き込んでいます。

アセンブリ言語では、アドレス指定をいろいろな方法で行なうことができますが、このようないろいろな方法のことをアドレッシング・モードと言い、16進数（オフセット・アドレス等）によって直接的にアドレスを指定する方法を直接アドレス指定（ダイレクト・アドレッシング）と、レジスタ等を介して間接的にアドレスを指定する方法を間接アドレス指定（インダイレクト・アドレッシング）と呼びます。

非常に多くのアドレッシング・モードが用意されていることが、808

の最大の特徴の1つとなっています。ここで、もう1つ、間接アドレス指定の例を示しておきましょう。

```
MOV     03H[BX+SI], 55H
```

上記例では、レジスタBXの内容にソース・インデックス・レジスタ(SI)の内容を加え、さらに03Hを加えたものをオフセット・アドレスとして、55Hを書き込みます。ここで、アドレス指定を示す大かっこの最前の03Hをディスプレイメントと呼びます。

ディスプレイメント

他にどのようなアドレッシング・モードが存在するのかを知りたい場合には、7.0節の表（メモリ・アドレッシング）をご参照ください。

0000 8CC8	MOV	AX,CS
0002 8ED8	MOV	DS,AX
0004 BB0010	MOV	BX,1000
0007 C60755	MOV	BYTE [BX],55
000A 43	INC	BX
000B C60755	MOV	BYTE [BX],55
000E 43	INC	BX
000F C60755	MOV	BYTE [BX],55
0012 43	INC	BX
0013 C60755	MOV	BYTE [BX],55
0016 43	INC	BX
0017 C60755	MOV	BYTE [BX],55
001A 43	INC	BX
001B C60755	MOV	BYTE [BX],55
001E 43	INC	BX
001F C60755	MOV	BYTE [BX],55
0022 43	INC	BX
0023 C60755	MOV	BYTE [BX],55
0026 43	INC	BX
0027 C60755	MOV	BYTE [BX],55
002A 43	INC	BX
002B C60755	MOV	BYTE [BX],55
002E 43	INC	BX
002F C60755	MOV	BYTE [BX],55
0032 43	INC	BX
0033 C60755	MOV	BYTE [BX],55
0036 43	INC	BX
0037 C60755	MOV	BYTE [BX],55
003A 43	INC	BX
003B C60755	MOV	BYTE [BX],55
003E 43	INC	BX
003F C60755	MOV	BYTE [BX],55
0042 F4	HLT	

5.1.1

AX ← CS

DS ← AX

BX ← 1000H

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

[BX] ← 55H

BX ← BX + 1

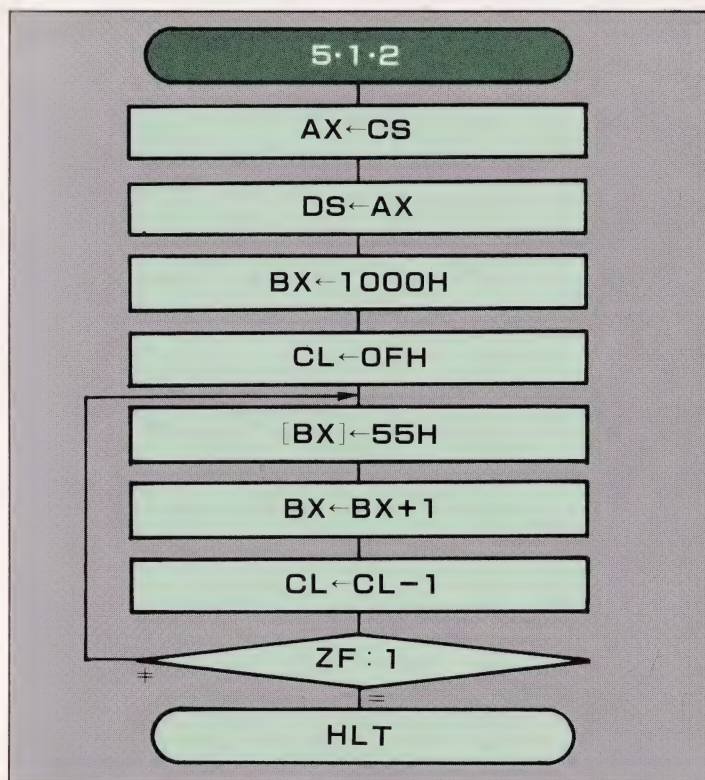
[BX] ← 55H

HLT

本節の場合には、55Hを書き込むメモリは15バイトのみですから反復構造を用いなくても何とかできましたが、これが100バイトや1000バイトの場合にはどうにもなりません。長大なプログラムを打ち込むことで、RAM空間も、プログラマ（キーパンチャ）の気力も尽きてしまうことは確実です。

そこで、メモリを指定するためのポインタ（レジスタBX）をインクリメントする部分と、ポインタで指定するメモリに55Hを書き込む部分を、反復構造によってくり返すように組み直しました。

0000 8CC8	MOV	AX,CS	← データ・セグメントをコード・セグメントに重ねる
0002 8ED8	MOV	DS,AX	
0004 BB0010	MOV	BX,1000	← ポインタに1000Hを設定する
0007 B10F	MOV	CL,0F	← ループ回数を15回を設定する
0009 C60755	MOV	BYTE [BX],55	← 55Hを書き込む
000C 43	INC	BX	← ポインタをインクリメントする
000D FEC9	DEC	CL	← ループカウンタをデクリメントする
000F 75F8	JNE	0009	← 15回に満たなければくり返す
0011 F4	HLT		← ホールト



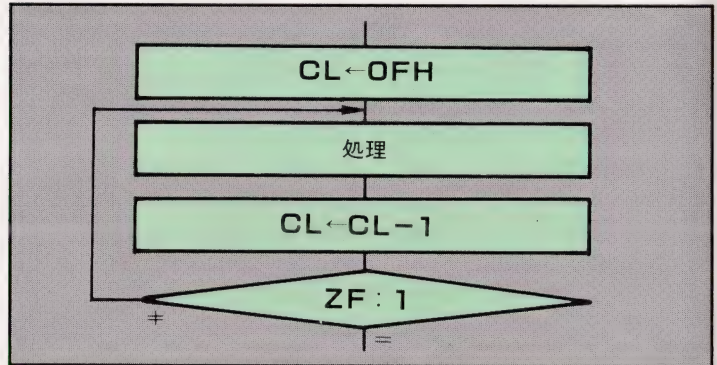
両者のフローチャートを比較することによって、反復構造にすることがいかに有効であったのかがわかると思います。

プログラム自体は非常にわかりやすいと思いますので特に説明ませんが、命令とフローチャートの中でループの制御に関係のあるものをぬき出して示します。くり返し処理の定石とも言えるものですので、ぜひやり方を憶えておいてください。

MOV CL, 0FH

DEC CL

JNE 0009H



ここでは、ループの処理回数をカウントするループ・カウンタとしてレジスタCLを用いていますが、処理を256回以上くり返す場合には、レジスタCXなど16ビットのレジスタを使用します。

```

hJc1800
hJa0000
0000 8CC8      mov ax,cs
0002 8ED8      mov ds,ax
0004 B80010     mov bx,1000
0007 B10F      mov cl,0f
0009 C60755     mov byte [bx],55
000C 43        inc bx
000D FEC9      dec cl
000F 75F8      jne 0009
0011 F4        hlt
0012

```

```

hJd1000,1010
1000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1010 00
hJg0000,0011
*1800:0011
hJd1000,1010
1000 55 55 55 55 55 55 55 55 55 55 55 55 55 00
1010 00
hJ

```

プログラムを入力する

メモリの内容を確認しておく

プログラムを実行する

結果を
確認する

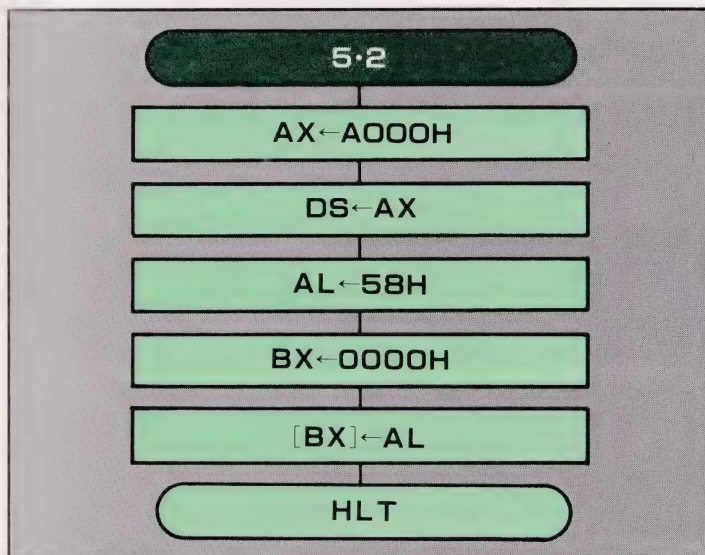
5.2 画面操作への応用——I

いま学んだループのつくり方を、実際の画面操作に応用できないで
しょうか？

PC-9801のVRAM構成については3.1節で述べました。

今回は、間接アドレッシングを用いて、VRAMの先頭アドレス(A0
000H番地)に58H(Xのキャラクタ・コード)を書き込んでみま
しょう。

さっそくフローチャートとプログラム・リストを示します。



0000 B800A0	MOV	AX, A000	} データ・セグメントをA0000H番地からに設定する
0003 8ED8	MOV	DS, AX	
0005 B058	MOV	AL, 58	} データ・セグメントの0000H番地に58Hを書き込む
0007 B80000	MOV	BX, 0000	
000A 8807	MOV	[BX], AL	
000C F4	HLT		← ホールト

あたりまえのことですが、このプログラムを実行することによって、
CRT画面のホーム・ポジション (LOCATE 0,0の位置) に、Xの文字
が表示されます。

プログラム中では、間接アドレッシングのためのポインタとして、レ
ジスタBXを用いていますが、その他にデータ(58H)の方も、一
度レジスタALに代入してからメモリに書き込んでいます。これは、
このプログラムを単にホーム・ポジションにXを表示するためだけのプ

プログラムとしてつくったのではなく、今後プログラムを発展させていろいろなことを行なうための基本形としてつくったためです。

本節から5.5節まで、フローチャートの形が少しずつ複雑になり、またプログラムの機能が少しずつ発展して行く様子を楽しんでいただきたいと思います。

アセンブリ言語の基礎を学ぶ過程で、たまにはCRT画面を操作してみるのもおもしろいものです。瞬間的にCRTに表示される実行結果をながめていると、時間を忘れてコンピュータ・ゲームに興じる人々の気持ちができるような気がします。

```
h]c1800
h]a0000
0000 B800A0      mov ax,a000
0003 8ED8        mov ds,ax
0005 8058        mov al,58
0007 BB0000      mov bx,0000
000A 8807        mov [bx],al
000C F4          hlt
000D
h]g0000,000c
*1800:000C
h]■
```

— プログラムを入力する

— プログラムを実行する

X

load auto go to list run save key print edit cont

5.3 画面操作への応用——II

次に示すプログラムは、CRT画面の最上段1列に80個のXを表示するプログラムです。

0000 B800A0	MOV	AX, A000	データ・セグメントをA0000H番地からに設定する
0003 8ED8	MOV	DS, AX	
0005 B058	MOV	AL, 58	表示するキャラクタをXに指定する
0007 B80000	MOV	BX, 0000	ポインタに0000Hを設定する
000A B150	MOV	CL, 50	ループ・カウンタに80を設定する
000C 8807	MOV	[BX], AL	ポインタが指定するアドレスにキャラクタを送る
000E 83C302	ADD	BX, 0002	ポインタに2を加える
0011 FEC9	DEC	CL	
0013 75F7	JNE	000C	ループ・カウンタをディクリメントしゼロでなければくり返す
0015 F4	HLT		ホールド

プログラムは、前節のものにループ処理を加え、VRAMにXのキャラクタ・コード(58H)を書き込んで行くことで80個のキャラクタを表示しており、基本的な反復構造を学ぶ上で非常に良い例になると思います。

アドレス指定用のポインタとしておなじみのレジスタBXを使い、ループ回数をカウントするためのカウンタとしては、これもおなじみのレジスタCLを使いました。したがって、ループ処理を1回くり返すことで、レジスタBXは加算命令

```
ADD BX, 0002H
```

によって、

```
0000H → 0002H → 0004H → ……
```

と増加し、逆にレジスタCLの値はディクリメント命令

```
DEC CL
```

によって、

```
50H → 4FH → 4EH → 4DH → …… → 00H
```

と減少して行きます。そして、レジスタCLの値が00Hになった時点の条件分岐命令

```
JNE 000CH
```

JNE(JNZ)

では、ゼロ・フラグ(ZF)が1にセットされているため、分岐せずにループから脱出します。

なぜVRAMを2番地おきに使い、奇数アドレスを空けておくのかはVRAMの構造を説明する時に述べましたが、このプログラムでは、ポインタ(レジスタBX)に2を加えるために加算命令を使っています。本来ならばインクリメント命令を2度続けて使い、


```

INC      BX
INC      BX

```

を実行した方が、機械語コードのバイト数も少なくなるのですが、1バイトのメモリを節約するよりもプログラムを見やすくする方を選びました。特に内蔵アセンブラ（モニタのAコマンド）を用いてプログラムを開発する場合には、画面上のリストを見ながらデバッグ等を行なうことが多いと思いますので、メモリ容量を減らすよりもリストの行数を減らすことを考えた方が得策です。

また、40桁モードの場合には、VRAMを4番地おきに使って、

0000H → 0004H → 0008H → ……

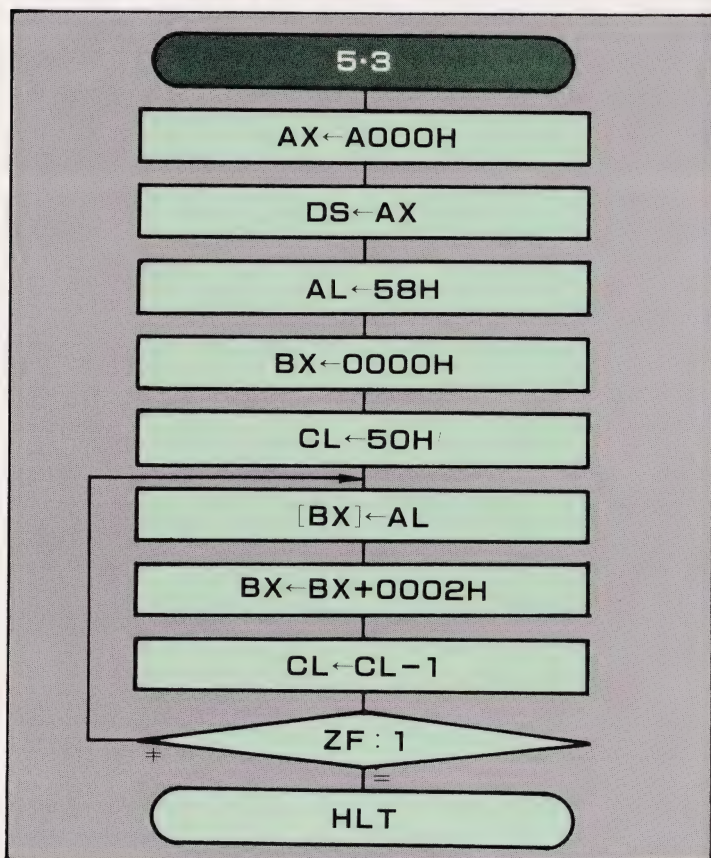
にXのキャラクタ・コード（58H）を入れて行けば良いわけですから、レジスタBXに2を加えるかわりに4を加える命令

```

ADD      BX, 0004H

```

を用いれば、40個表示した時点で自動的に次の行に進み、80個のXを2行に分けて表示します。



```

h]c1800
h]a0000
0000 B800A0      mov ax,a000
0003 8ED8        mov ds,ax
0005 B058        mov al,58
0007 B80000      mov bx,0000
000A B150        mov cl,50
000C 8807        mov [bx],al
000E 83C302      add bx,0002
0011 FEC9        dec cl
0013 75F7        jne 000c
0015 F4          hlt
0016
h]g0000,0015
*1800:0015
h]■

```

プログラムを入力する

プログラムを実行する

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

load auto go to list run save key print edit cont

5.4 画面操作への応用——III

今度は1行だけでなく、CRT画面全体をXの文字で埋めてみましょう。
つまり、2000個（80桁×25行）のXを表示することになります。
プログラム・リストを示します。

0000 B800A0	MOV	AX,A000	←データ・セグメントをA0000H番地からに設定する
0003 8ED8	MOV	DS,AX	
0005 B058	MOV	AL,58	←表示するキャラクタをXに指定する
0007 BB0000	MOV	BX,0000	←ポインタに0000Hを設定する
000A B519	MOV	CH,19	←ループ・カウンタに25行を設定する
000C B150	MOV	CL,50	←ループ・カウンタに80桁を設定する
000E 8807	MOV	[BX],AL	←ポインタが指定するアドレスにキャラクタを送る
0010 83C302	ADD	BX,0002	←ポインタに2を加える
0013 FEC9	DEC	CL	←ループ・カウンタをディクリメントしゼロでなければくり返す
0015 75F7	JNE	000E	
0017 FECD	DEC	CH	←ループ・カウンタをディクリメントしゼロでなければくり返す
0019 75F1	JNE	000C	
001B F4	HLT		←ホールド

リストまたは142ページのフローチャートを見ていただいてもわかるように、このプログラムではループが2重になっています。これをネスティング・ループなどと言い、ここでは2重ですが、場合によっては3重4重以上のネスティングを行なうこともあります。

ネスティング

MOV CH, 19H

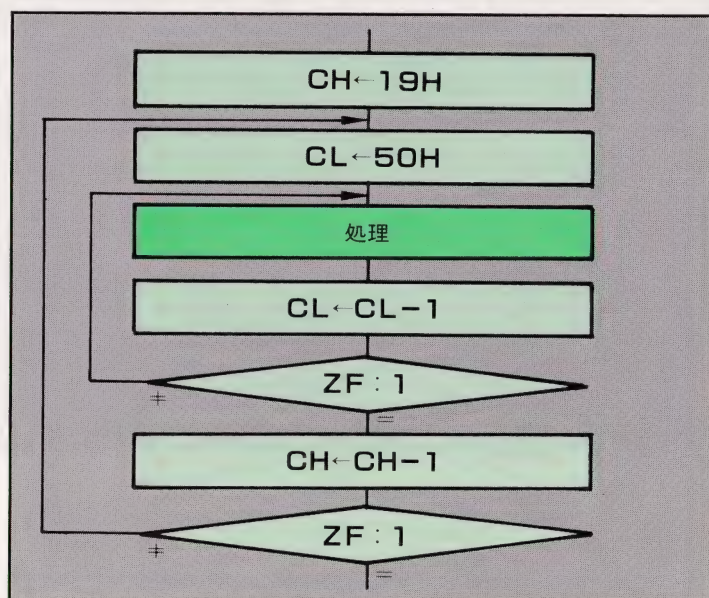
MOV CL, 50H

DEC CL

JNE 000EH

DEC CH

JNE 000CH

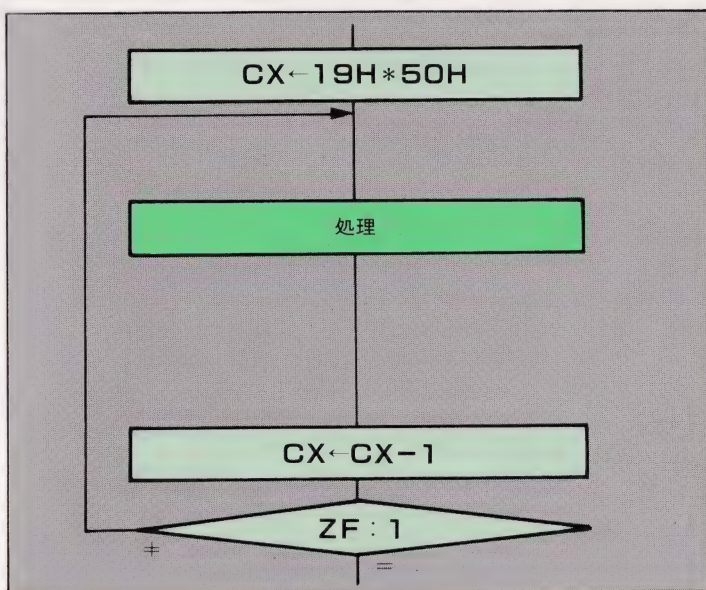


このプログラムでは、単に画面を埋めるだけですから、ネスティング・グループを行なう必然性は全くなく、2000回の1重ループですませた方が自然です。

しかし、80桁*25行というフィールド等を埋めて行くような場合には、1行分の作業を終えた時点で何らかの処理を加えることが多いものです。PC-9801のテキストVRAMの場合には、前の行の末尾と次の行の先頭がつながっているため改行することを意識する必要はありませんが、そうでない場合には1行分の作業を終えるごとに、ポインタ（レジスタBX）に次行の先頭アドレスをセットし直さなければなりません。このような理由からあえてネスティングを行ないました。

ご参考にといい、同じ処理をネスティングを行なわないように作り直したプログラム・リストもここに掲げておきます。

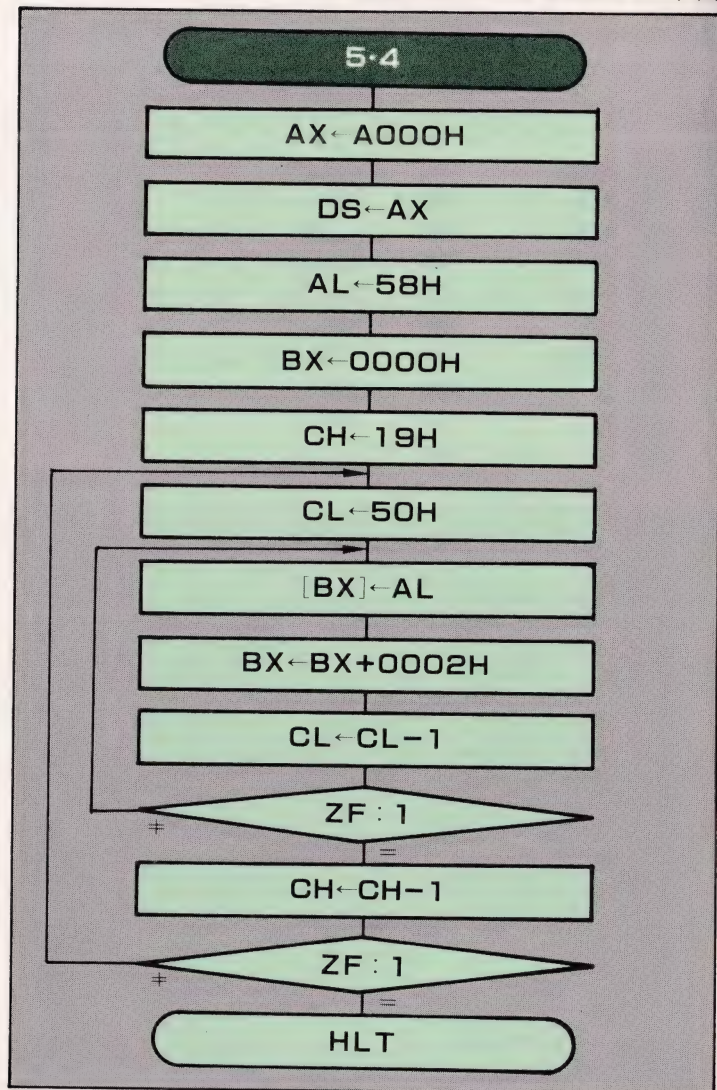
0000 B800A0	MOV	AX, A000	データ・セグメントをコード・セグメントに重ねる
0003 8ED8	MOV	DS, AX	
0005 B058	MOV	AL, 58	表示するキャラクタをXに指定する
0007 B80000	MOV	BX, 0000	ポインタを0000Hに設定する
000A B9D007	MOV	CX, 07D0	ループ・カウンタを80*25に設定する
000D 8807	MOV	[BX], AL	ポインタで指定するアドレスにキャラクタを送る
000F 83C302	ADD	BX, 0002	ポインタに2を加える
0012 49	DEC	CX	ループ・カウンタをデクリメントしゼロでなければくり返す
0013 75F8	JNE	000D	
0015 F4	HLT		ホールド



プログラムを実行すると、一瞬にしてCRT画面中がXのキャラクタで埋まりますが、実際には、最上行の左端から右端へ80個のXを表示して行き、それを最下段までの25行分くり返すことによって2000文字のXを書いていることを忘れてはいけません。

N88-BASIC(86)等によって、同じアルゴリズムのプログラムをつくり実行してみると、画面が埋められて行く状態がゆっくりとシュミレートされておもしろいかも知れません。

また、本節のプログラムでは画面を埋めるキャラクタとしてXを用いましたが、たとえばスペース(空白=20H)を埋めることで、画



■ リアなどに応用することができます。

次にプログラムの実行手順および実行結果の画面コピーを示します。

```
h3c1800
h3a0000
0000 B800A0          mov ax,a000
0003 8ED8             mov ds,ax
0005 B058             mov al,58
0007 B80000           mov bx,0000
000A B519             mov ch,19
000C B150             mov cl,50
000E 8807             mov [bx],al
0010 83C302           add bx,0002
0013 FEC9             dec cl
0015 75F7             jne 000e
0017 FEC0             dec ch
0019 75F1             jne 000c
001B F4              hlt
001C
h3g0000,001b
#1800:001B
h3H
```

プログラムを入力する

プログラムを実行する

A large rectangular area filled with a dense, repeating pattern of small, stylized, light-colored symbols (possibly 'X' or 'O') on a dark background, resembling a textured surface or a data visualization.

load auto go to list run save key print edit cont

5.5 画面操作への応用——IV

同じキャラクタ(X)ばかり表示させていても面白くありませんので、今度は表示するキャラクタを変化させてみました。

前節までのプログラムでも、Xのキャラクタ・コードである58HをレジスタALに格納しておき、ポインタ（レジスタBX）の指すアドレスにレジスタALの値を移すという2段構えで、VRAMにキャラクタ・コードを送り込んでいますから、ループ中でレジスタALの値を変化させることによって、簡単に表示するキャラクタを変更することができます。

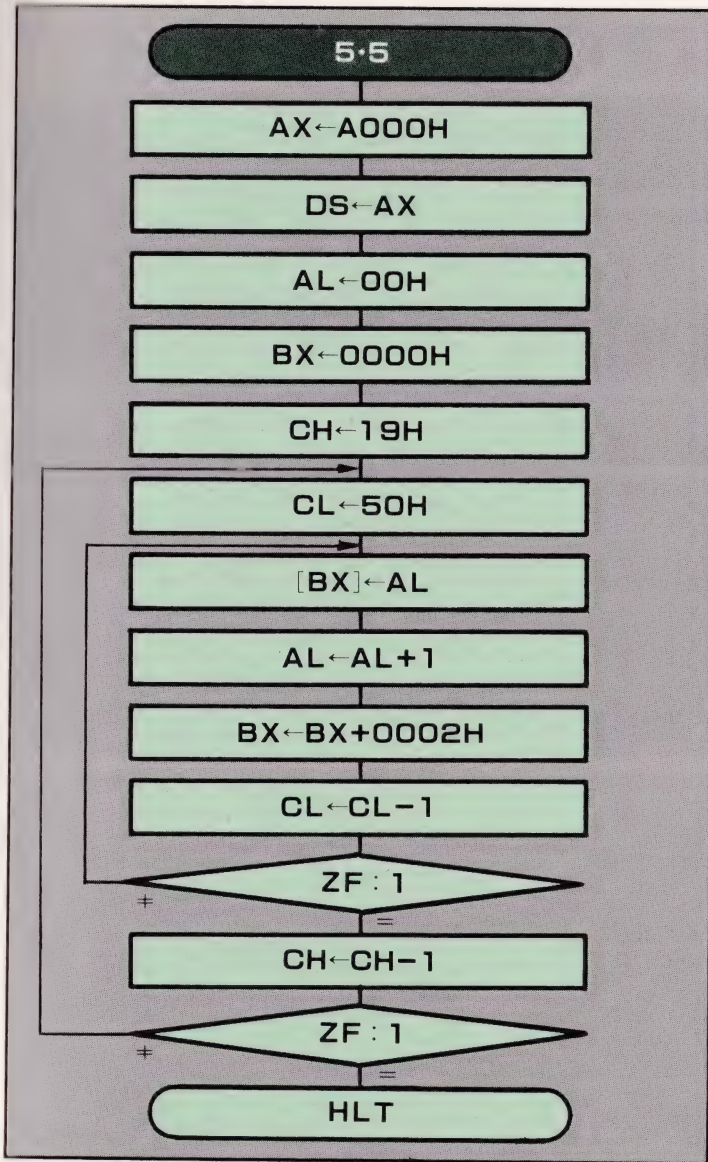
プログラム・リストおよび次ページにフローチャートを示します。

0000 B800A0	MOV	AX,A000	データ・セグメントをA0000H番地からに設定する
0003 8ED8	MOV	DS,AX	
0005 B000	MOV	AL,00	表示するキャラクタを初期化する
0007 B80000	MOV	BX,0000	ポインタに0000Hを設定する
000A B519	MOV	CH,19	行動用のループ・カウンタに19Hを設定する
000C B150	MOV	CL,50	桁動用のループ・カウンタに50Hを設定する
000E 8807	MOV	[BX],AL	ポインタが指定するアドレスにキャラクタ・コードを送る
0010 FEC0	INC	AL	表示するキャラクタを変更する
0012 83C302	ADD	BX,0002	ポインタに2を加える
0015 FEC9	DEC	CL	ループ・カウンタをデクリメントしゼロでなければくり返す
0017 75F5	JNE	000E	
0019 FEC0	DEC	CH	ループ・カウンタをデクリメントしゼロでなければくり返す
001B 75EF	JNE	000C	
001D F4	HLT		ホールド

レジスタALには、最初00Hを入れておき、最小のループすなわち1つのキャラクタを表示するごとにインクリメントしています。レジスタALは1バイトのレジスタですから、FFHの後はインクリメントされて00Hにもどり、このため、キャラクタ・コードが00HのキャラクタからFFHのキャラクタを画面が埋まるまでくり返して表示するようになっていきます。

ここで注目したいのが00Hから1FHまでのコントロール・コードです。

普通、BASICの命令（PRINT等）によってキャラクタを表示するような場合には、コントロール・キャラクタを表示することができません。いや、不可能なわけではなく、コントロール・キャラクタは特別な機能を持っているため、CRT画面をクリアしてしまったり、カーソルを移動したりするために使用するキャラクタだからです。



しかし、直接VRAMにコードを書き込む場合には、コントロール・キャラクタも他のキャラクタと同じように表示されます。

これは、コントロール・コードとしての機能は、N88-BASIC (86) 内部 (実際にはiN88-BASIC のBIOS) の表示ルーチンを通ることで はじめて発揮できることを意味し、もし機械語でそれらのコントロール機能を使いたければ、基本的には処理ルーチン全てを、プログラマ がサポートしなければなりません。

BIOS

●1バイト入出力等のハードウェアに密接した基本的なI/O処理ルーチンをまとめた部分

実行例を示します。ただし、コントロール・キャラクタは、プリンタの関係で印字されていません。

```
hJc1800
hJa0000
0000 8800A0      mov ax,a000
0003 8ED8        mov ds,ax
0005 B000        mov al,00
0007 BB0000      mov bx,0000
000A B519        mov ch,19
000C B150        mov cl,50
000E 8807        mov [bx],al
0010 FEC0        inc al
0012 83C302      add bx,0002
0015 FEC9        dec cl
0017 75F5        jne 000e
0019 FEC0        dec ch
001B 75EF        jne 000c
001D F4         hlt
001E
hJg0000,001d
*1800:001D
hJH
```

[illegible]

5.6 簡単なデモンストレーション

画面操作および反復構造の締めくくりにして、簡単なデモンストレーション・プログラムをつくってみましょう。

デモンストレーションは、前ページ下図（前節の実行画面）の画面が永久的に横スクロールをくり返すものです。少し古いですが、回りの龍のコンピュータ版といったところでしょうか！

プログラム自体も前節のプログラム・リストに新しい3ステップの命令を付け加えただけで、いかにして横スクロールを行なうかではなく、いかにして横スクロールを行なっているように見せるかを考えてみました。

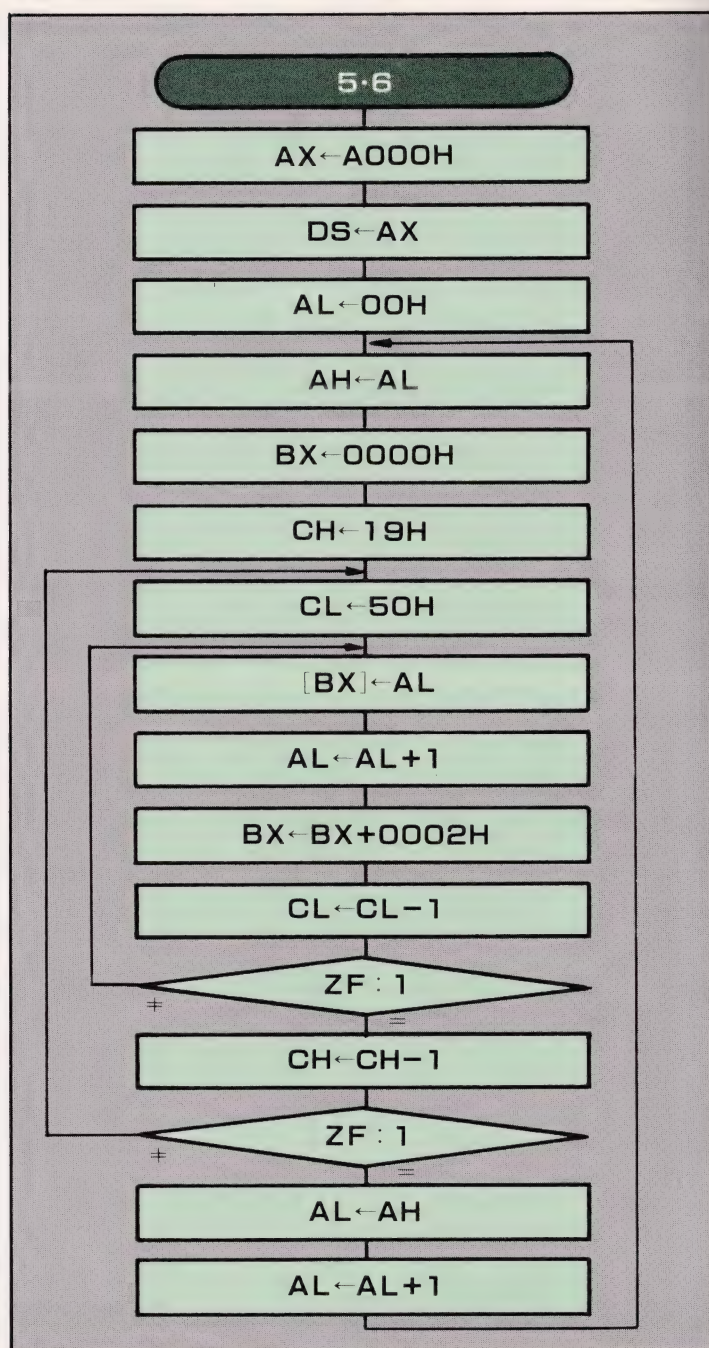
このプログラムは、ぜひPC-9801に入力して走らせてください。そして、プログラム・リストが、ただの17ステップの命令でしか構成されていないことも思い出してください。機械語のすばらしいこと、また機械語が少しも手間のかかるものではないことにも、きっと気づいていただけたと思います。

話しが少し大げさになってしまいました。

さて、プログラムですが、フローチャートを見るとわかるように、3重のネスティング・ループになっています。しかも、最も外側のループが永久にくり返す無限ループになっていますので、デモンストレーションを止める時には **STOP** キーを押しながらリセット・スイッチを押してください。

0000 B800A0	MOV	AX,A000	データ・セグメントをA0000H番地からに設定する
0003 8ED8	MOV	DS,AX	
0005 B000	MOV	AL,00	表示するキャラクタを初期化する
0007 88C4	MOV	AH,AL	キャラクタをレジスタAHに退避する
0009 BB0000	MOV	BX,0000	ポインタに0000Hを設定する
000C B519	MOV	CH,19	行番号のループ・カウンタに19Hを設定する
000E B150	MOV	CL,50	桁数用のループ・カウンタに50Hを設定する
0010 8807	MOV	[BX],AL	ポインタが指定するアドレスにキャラクタ・コードを送る
0012 FEC0	INC	AL	表示するキャラクタを変更する
0014 83C302	ADD	BX,0002	ポインタに2を加える
0017 FEC9	DEC	CL	ループ・カウンタをディクリメントしゼロでなければくり返す
0019 75F5	JNE	0010	
001B FEC0	DEC	CH	ループ・カウンタをディクリメントしゼロでなければくり返す
001D 75EF	JNE	000E	
001F 88E0	MOV	AL,AH	横スクロールに見せるためキャラクタの初期値を変更する
0021 FEC0	INC	AL	
0023 E9E1FF	JMP	0007	くり返す

最後にフローチャートを示し、画面操作への応用を締めくくりたい
と思います。



hJc1800	
hJa0000	
0000 B800A0	mov ax,a000
0003 8ED8	mov ds,ax
0005 B000	mov al,00
0007 88C4	mov ah,al
0009 B80000	mov bx,0000
000C B519	mov ch,19
000E B150	mov cl,50
0010 8807	mov [bx],al
0012 FEC0	inc al
0014 83C302	add bx,0002
0017 FEC9	dec cl
0019 75F5	jne 0010
001B FEC0	dec ch
001D 75EF	jne 000E
001F 88E0	mov al,ah
0021 FEC0	inc al
0023 E9E1FF	jmp 0007
0026	
hJg0000 ←	プログラムを実行する

プログラムを入力する

このプログラムに関しては、あえて実行結果
を示しませんでした。

PC-9801

に入力して走らせてください！

5・7 ローテートおよびシフトの概念

ローテート・シフト命令とは、何を行なうための命令なのでしょう？ 本節では、2進数を扱って行く上で非常に重要な、ローテート(回転) およびシフトについて少し詳しく述べてみましょう。

ローテート・シフトの概念は、BASIC等の高級言語にはほとんどあらわれない、アセンブリ言語独特のもので、いろいろな応用を行なう上で必ず必要となってくる考え方です。2進数の表現方法さえ理解していれば、命令の動作自体は単純なのですが、たいへん底の深い命令であることは確かです。

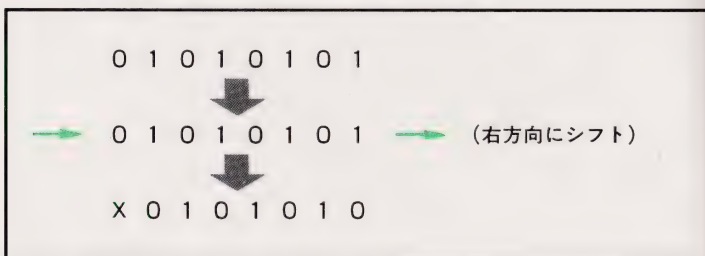
皆さんもよくご存知のように、メモリ上のデータは全て2進数で記憶されています。この2進数のデータが8ビット集まって1バイトを構成しているのですが、人間の側から見るとコンピュータのメモリ上にも、55Hなどのように16進数で記憶されているように思えます。もっとも、高級言語のみを使っている場合には、コンピュータの中にも10進数や文字などが直接記憶されていると思って不思議ではありません。まるで写真のフィルムでも入っていて、CRT画面に投影されているような感じさえ受けてしまいます。

少し話題からそれましたが、1バイトの16進数も実際には8ビットの2進数であることを強調したかったのです。

例をあげれば、55Hは実際には01010101Bであり、AAHであれば10101010Bなのです。

シフト命令

このような2進数の列を左右にずらす(シフトする)ための命令がシフト命令です。先ほどの01010101B(55H)を右方向にシフトしてみましょう。



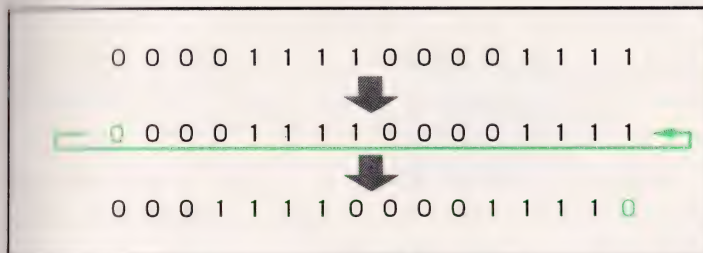
右方向にシフトすることによって、最下位ビット(LSB)が消えた分、全てのビットが右にずれ、最上位ビット(MSB)に何かが入ってきたのがわかります。このXの部分に何が入るのかは、各種の命令が用意されていて、各々異なります。

ローテート命令

ローテート命令の場合には、2進数の列の両端をつないで輪のように

これらのものをローテート（回転）させるのが目的です。

今度は、16ビットのデータ（O F O F H）を左まわりにローテートしてみましょう。



それでは、2進数を左右にローテートまたはシフトした場合の意義は何でしょうか？ 少し考えてみましょう。

皆さんが、10進数で表現されている数値を、10倍、100倍、1000倍…したい場合にはどのようにしますか？

まさか、1234を1000倍するために、右図のような筆算を用いる人はいないと思います。普通は、1234の末尾に3個の0を付け加えれば、簡単に解答が求められます。逆に、10分の1、100分の1、1000分の1……、も、小数点の位置を左に移動するだけで求められるわけです。

$$\begin{array}{r}
 1234 \\
 *) 1000 \\
 \hline
 0000 \\
 0000 \\
 0000 \\
 1234 \\
 \hline
 1234000
 \end{array}$$

以上は10進数の場合ですが、実際にコンピュータで扱う2進数の場合には、どうなるのかをためてみましょう。

8ビット長の2進数0 0 0 0 0 0 0 1 B（10進表現では1）を基準にして、最下位に0を付け加え、最上位ビットを削ってみました。つまり、左にシフトしているのと同じことになります。

0 0 0 0 0 0 0 1		1
0 0 0 0 0 0 1 0		2
0 0 0 0 0 1 0 0		4
0 0 0 0 1 0 0 0		8
0 0 0 1 0 0 0 0		16

右側の10進数を見てもわかるように、最初の数値に比較して、2倍、4倍、8倍、16倍……、となっています。

逆に、右にシフトする事で、2分の1、4分の1、8分の1、16分の1……、が求められることも容易に想像がつくと思います。

このように、2進数のデータをnビットだけシフトすることは、小数点の位置をn個分移動することであり、難しく言えば 2^n を乗じることでもあります。

高度なプログラムでは、この性質を巧みに利用することが必要です。

5・8 ローテート・シフト命令を使った乗算方法

さっそく、ローテート・シフト命令を使用したプログラムをつくってみましょう。どのようなプログラムにするのかを考えましたが、ローテート・シフト命令の使用例として最も一般的な、8ビット乗算プログラムをつくることにしました。

8086には、簡単に乗除算を行なうことのできる専用の命令が用意されていますが、ローテート・シフト命令による乗算技術は、アセンブリ言語を学ぶ上で非常に重要な技法だと思います。

また、一瞬で答を算出しているように見えるコンピュータでも、実際には人間が筆算を行なう時のように1桁ずつ計算して行く、8086の人間らしい(?)一面を知ることができると思います。

さて、乗算命令を使わずに乗算を行なうアルゴリズムは、どのようなものになるのでしょうか？

ために、 13×12 の演算を筆算で行なったものを、10進数の場合と2進数の場合に分けて欄外に示します。

```
    13
*) 12
  ---
   26
  13
 ---
 156
```

```
    00001100
*) 00001101
  ---
 00001100
 00000000
 001100
 01100
 0000
 000
 00
 0
 ---
10011100
```

両者を比較していただければおわかりのように、10進数の筆算を行なうためには、最低でも、多桁×1桁の乗算を行なうことが不可欠であり、どのように工夫しても乗算からのがれることはできません。私達が小学校の算数の授業で、九九算を記憶しなければならなかったのもこのような理由によるものなのでしょう。

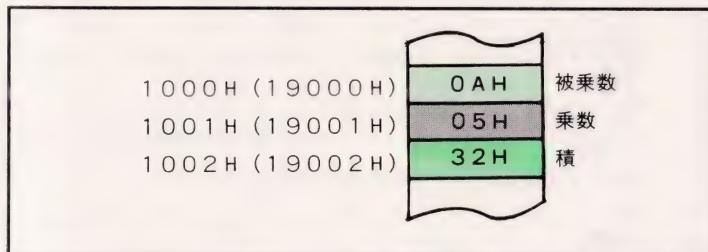
これに反して、2進数で筆算を行なう場合には、乗算は全く必要ありません。なぜならば、2進数では、「0を乗じる」か「1を乗じる」かだけなので、結果を「0」と「もとの値」のどちらかに限定することができるからです。

この、2進数の筆算アルゴリズムを、プログラムでシミュレートすることが、ローテート・シフト命令による乗算プログラムに他なりません。

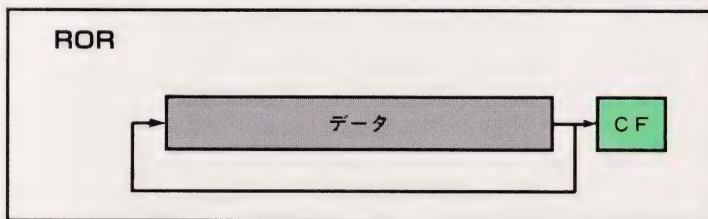
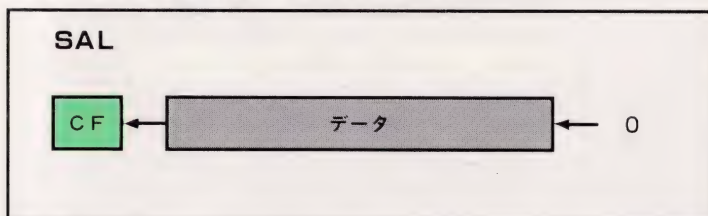
ようするに、乗数を最下位のビットから順に調べて行くと同時に、被乗数を左にシフト(2倍)して行き、調べた乗数のビットが1であった場合のみ、シフトして来た被乗数を積に加えて行けば良いわけです。文章ではわかりにくいかも知れませんが、アルゴリズムを簡潔書きにしてみます。

- I 積を0にクリアします。
- II 乗数を右方向にローテートし最下位ビットを調べます。
- III 調べたビットが1であれば積に被乗数を加えます。
- IV 被乗数を1ビット分左方向にシフトします。
- V IIからIVまでを8回くり返します。

以上のアルゴリズムによってプログラムをつくりませんが、例によって、被乗数、乗数および積を格納するためにメモリを割り当てておきましょう。



さらに、上記の手順IIおよび手順IVで2種のローテート・シフト命令、すなわちRORおよびSALを使いますので、それぞれの命令が第1オペランドにほどこす操作を図示しておきます。



つまり、SALを実行することによりデータを左方向にシフトすることができます。ここでは、コード・セグメントの1000H番地に格納された被乗数をシフトするために、

`SAL BYTE [1000H], 01H`

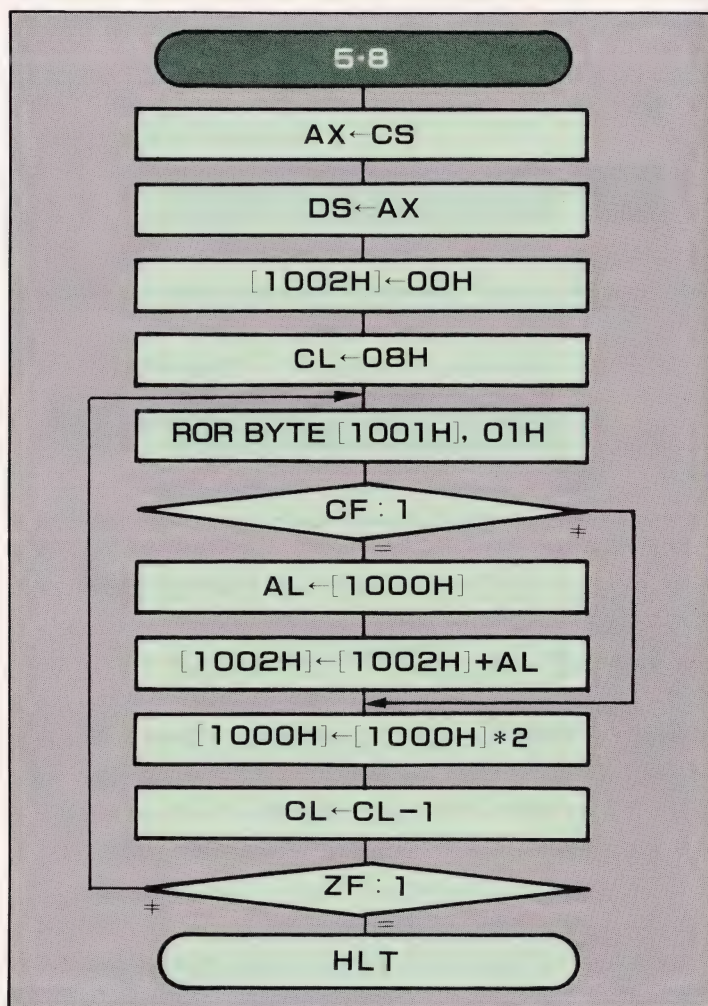
を用いています。オペランドの先頭にあるBYTEはデータが1000H番地のみに格納されたバイト・データであることを示し、第2オペランドの01Hはシフトするビット数が1ビットであることを示します。さらに8086のローテート・シフト命令では、ローテートまたはシフトするビット数をレジスタCLを第2オペランドに置いて指定することもできます。

RORは、データを右方向にローテートすると同時に、最下位ビットにあったデータ(0または1)をキャリ・フラグ(CF)に設定する命令ですが、ここでは1001H番地の乗数を最下位ビットから順に調

SAL

ROR

0000 8CC8	MOV	AX,CS	
0002 8ED8	MOV	DS,AX	データ・セグメントをコード・セグメントに重ねる
0004 C606021000	MOV	BYTE [1002],00	(I) 積を0にクリアする
0009 B108	MOV	CL,08	ループ・カウンタを8回にセットする
000B D00E0110	ROR	BYTE [1001],01	(II) 乗数を右方向にローテートし積下位ビットを調べる
000F 7307	JAE	0018	
0011 A00010	MOV	AL,[1000]	(III) 調べるビットが1であれば積に乗数を加える
0014 00060210	ADD	[1002],AL	
0018 D0260010	SAL	BYTE [1000],01	(IV) 乗数を1ビット分左方向にシフトする
001C FEC9	DEC	CL	
001E 75EB	JNE	000B	ループ・カウンタをディクリメントしゼロでなければくり出す
0020 F4	HLT		ホールド



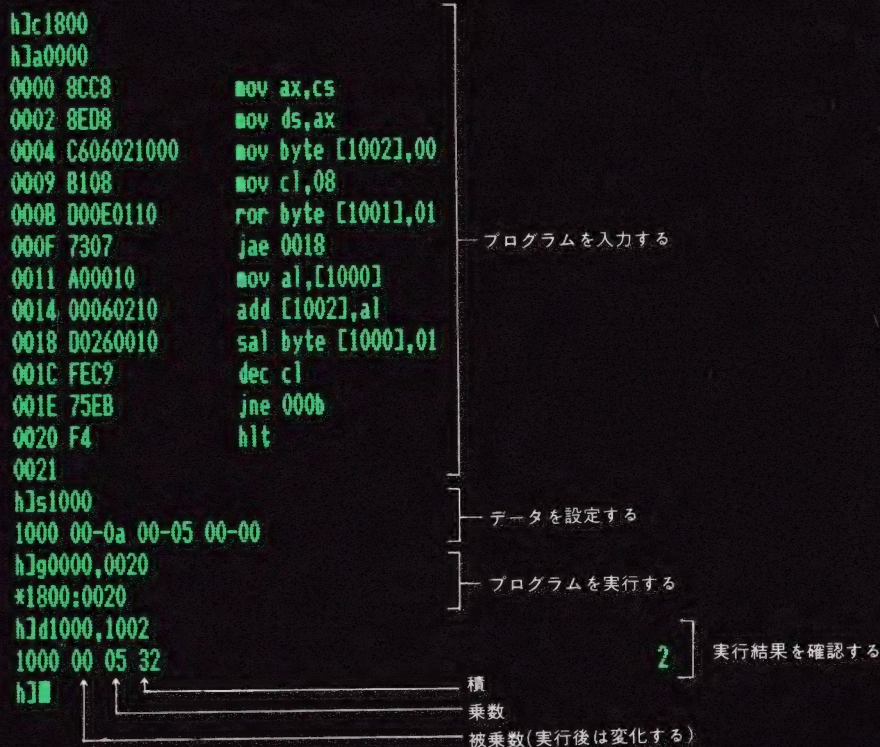
べて行くために、

ROR BYTE [1001H], 01H

を、キャリ・フラグ (CF) による条件分岐命令と組み合わせて利用しています。

前ページに、8ビット乗算プログラムのプログラム・リストおよびフローチャートを示しました。

選択構造と反復構造が組み合わさった少し複雑なプログラムなので、多少理解しにくいかも知れませんが、フローチャートとともにニーモニックを追ってみてください。また、余力のある方は、16ビットの乗算アルゴリズムや、除算アルゴリズムなどを考えるのもおもしろいかも知れません。これらのアルゴリズムを考え出して行くことが、まさにアセンブリ言語の楽しさであり、アセンブリ言語に限らずシステム設計、プログラミング、コーディングなどに携わる際の格好の訓練になるであろうと思います。



5・9 割り込み(インタラプト)の基礎知識

割り込み(インタラプト)の考え方は、コンピュータを使いこなす上で非常に重要な考え方です。本節では、8086の持つ割り込み機能について、ソフトウェアの面から簡単に解説することにしましょう。

一般に割り込みというのは、CPU外部の外部機器からの呼び出しを受けるためのメカニズムで、NMI (NON MASKABLE INTERRUPT) とINTR (MASKABLE INTERRUPT) が設けられています。前者のNMIは絶対にマスク(禁止)することのできない割り込みであるため、たとえばハードウェアのトラブルなど、緊急を要する場合にのみ利用します。これに対してINTRは、重要な作業の途中などで割り込みを受け付けたくない場合にはマスク(禁止)しておくこともできます。具体的には、インタラプト・イネーブル・フラグ(IF)を0にクリアしておくことでINTRを受け付けず、1にセットしておくことでINTRを受け付けます。

インタラプト・イネーブル・フラグ(IF)をクリアする命令はCLIで、セットする命令はSTIです。

さて、これらの割り込みを受け付けたCPUは、フラグ(F)、コード・セグメント・レジスタ(CS)、命令ポインタ(IP)の値をスタックに退避した後、普通割り込み処理ルーチンなどと呼ぶサブルーチンに分岐します。割り込み処理ルーチンの最後には、IRETを置くことによってメイン・ルーチンへもどることができます。

8086には、タイプ0からタイプ255まで計256種類の割り込みを区別して受け付けることができますが、このため最大256か所の割り込み処理ルーチンを用意することが必要となります。したがって、256か所のルーチンへ分岐するために、各ルーチンの先頭アドレスを明確にしておかなければなりません。8086では00000H番地からの $256 \times 4 = 1024$ バイトを、各割り込み処理ルーチンの先頭アドレスを格納しておくためのテーブル(割り込みベクタ・テーブルと呼ぶ)として使用します。

ここで、1種類の割り込みに対して、セグメント・ベース・アドレスとオフセット・アドレスの両方が必要なため、4バイトずつのエリアが確保されている点に注意してください。

たとえば、タイプ0の割り込みに対する割り込み処理ルーチンの先頭アドレスは、00000H~00001H番地にオフセット・アドレスが、00002H~00003H番地にセグメント・ベースアドレスが格納されていることになります。この様子を次図に示します。

NMI
INTR

CLI
STI

IRET

割り込みベクタ・テーブル

00000H	29H	タイプ0に対するオフセット・アドレスの下位バイト
00001H	04H	タイプ0に対するオフセット・アドレスの上位バイト
00002H	80H	タイプ0に対するセグメント・ベース・アドレスの下位バイト
00003H	FDH	タイプ0に対するセグメント・ベース・アドレスの上位バイト
00004H	29H	タイプ1に対するオフセット・アドレスの下位バイト
003FEH	80H	タイプ255に対するセグメント・ベース・アドレスの下位バイト
003FFH	FDH	タイプ255に対するセグメント・ベース・アドレスの上位バイト

割り込みベクタ・テーブルの内容が上記例のような状態である場合、タイプ0の割り込みを受け付けた8086は、FD800H番地から64Kバイトのセグメント中、オフセット・アドレスが0429H番地の割り込み処理ルーチン (FD800H+0429H=FDC29H番地) に分岐します。

ところで、今まで説明して来たことはハードウェアに起因することであって、ソフトウェア中心に考えている私達にとってはあまり重要なことではありません。私達にとって本当に重要な点は、これらの割り込み処理ルーチンの中には非常に有用なルーチンが多いということです。たとえば、文字や文字列をCRT画面やプリンタに出力したり、キーボードから入力したりする作業は最も基本的かつ重要な作業の1つと言えるもので、これらの基本的なサブルーチンは割り込み処理ルーチンとして用意されているはずです。

そこで、割り込み処理ルーチンを一般のサブルーチンと同じように呼び出す (プログラム中で割り込みを発生する) ための命令があると便利ですが、このための命令がINTです。

INT

INTはオペランド (0~255) によって割り込みのタイプを指定することができます。たとえば、上記例のFDC29H番地に存在する割り込み処理ルーチンを実行したければ、次の命令を使います。

INT 00H

せっかく便利な命令 (INT) が存在するにもかかわらず、割り込み処理ルーチンの内容が公表されていないのは非常に残念なことです。私自身も少しずつ調べてはいるのですが、90Kバイト以上もある N88-BASIC(86)を調べ尽くすのは個人では不可能かも知れません。ある程度まとまった時点で、また発表しようかとも思っています。

5・10 本文を読み終えた皆さんに

「アセンブリ言語プログラミング入門Ⅰ」の本文も最後になってしまいました。

入門書を書くというのは思っていた以上にむずかしいもので、はじめは十分だと思っていた200ページあまりの紙面でも、本当に基礎の基礎にしか触れることができませんでした。このような意味もあり、本書のタイトルに、あえて時計数字の「Ⅰ」を付けさせていただきます。「基礎編」または「STEP Ⅰ」などと受け取っていただければ幸いです。

アセンブリ言語に限らず、コンピュータを学ぶ上で最も重要なことは「いかにコーディングを行なうか」ではなくて「いかにアルゴリズムを組み立てるか」であり、まして「いかに多くの命令を覚えるのか」では絶対にありません。

そして、この、アルゴリズムを組み立てる能力と、組み立てたアルゴリズムを命令の組み合わせに置き換える能力は、実際にプログラミングを手がけることによってのみ向上すると私は思います。

皆さんが、本書で学んだ基礎知識やサンプル・プログラムを軸として、より高度な段階に進まれることを願ってやみません。そのような場合にこそ、次の第6章および最後の章が参考になるであろうと思います。

最後に、本書の執筆にあたって参考させていただいた文献類を示します。どの文献も非常にすぐれたものばかりです。

- ① Stephen P. Morse 『The 8086 Primer』

HAYDEN BOOK COMPANY, INC.

- ② 『MCS-86 User's Manual』

Intel Corp.

- ③ ステファン P. モールス 内藤祥訳 『8086入門』

株式会社システムソフト福岡

- ④ 『iAPX86ファミリー・ユーザーズマニュアル』

- ⑤ 『iAPX86マクロアセンブリ言語プログラミングマニュアル』

インテルジャパン株式会社

- ⑥ 『PC-9801 ユーザーズ・マニュアル』

- ⑦ 『PC-9801 BASICリファレンス・マニュアル』

日本電気株式会社

- ⑧ 脇英世著 『PC-8801 アセンブリ言語プログラミング入門』

株式会社ナツメ社

6章 | 8086の
インストラクション・セット



6.0 8086のインストラクション・セット

本章は、8086のアセンブリ言語が有するニーモニック・セットの全般にわたり、PC-9801のハードウェアにとらわれることなく解説を行なうものです。

したがって、前章までを読んで来られたことによって8086のアセンブリ言語によるプログラミングの基礎を学ばれた皆さんはもちろんのこと、他のCPUに関する知識をお持ちの方が8086のインストラクション・セットを評価されたい場合などにも十分に役立つよう、一読するだけで、全てのインストラクションを把握できるように考慮の上、執筆いたしました。

紙面の都合等もあるため突っ込んだ解説は不可能ですが、数多いインストラクションそれぞれの有用性をわかっていたいただければ幸いです。

分類			
1 データ転送命令	◎汎用データ転送命令	MOV	PUSH
	◎アキュムレータ転送命令	IN	OUT
	◎アドレス・オブジェクト転送命令	LEA	LDS
	◎フラグ・データ転送命令	LAHF	SAHF
2 算術演算命令	◎加算命令	ADD	ADC
	◎減算命令	SUB	SBB
3 乗除算命令	◎乗算命令	MUL	IMUL
	◎除算命令	DIV	IDIV
4 論理演算命令	◎シングル・オペランド論理命令	NOT	SHL
		RCR	
	◎ダブル・オペランド論理命令	AND	TEST
5 スtring操作命令	◎リピート・プリフィクス命令	REP	REPE
	◎プリミティブ・String命令	MOVS	MOVSB
6 分岐命令	◎制御転送命令	CALL	JMP
	◎条件付制御転送命令	JE	JZ
		JBE	JNA
		JNL	JGE
		JNP	JPO
	◎くり返し転送命令	LOOP	LOOPZ
7 CPU制御命令	◎割り込み転送命令	INT	INTO
	◎フラグ制御命令	CLC	CMC
	◎CPU制御命令	HLT	WAIT

各インストラクションは、巻末付録のインストラクション活用表に準じて分類いたしました。この巻末付録は、インテル社などの分類方法とも一致しています。多少の無理も見受けられますが、最も標準的な分類方法と言えましょう。

また、できる限り、分類された順序に添って紹介し、全てのニーモニックを欄外に抜き出しておきました。ニーモニックを見出しとしてインストラクションを調べる機会なども多々あるかと思いますので、そのような場合にも、小辞典として手軽にご利用ください。

なお、N88-BASIC (86) 内蔵モニタのAコマンドではサポートされていない、一部のインストラクションも存在しますので、ご注意ください。

ニーモニック

POP	XCHG				
XLAT					
LES					
PUSHF	POPF				
INC	AAA	DAA			
DEC	NEC	CMP	AAS	DAS	
AAM					
AAD	CBW	CWD			
SAL	SHR	SAR	ROL	ROR	RCL
OR	XOR				
REPZ	REPNE	REPNZ			
MOVSW	CMPS	SCAS	LODS	STOS	
RET					
JL	JNGE	JLE	JNG	JB	JNAE
JP	JPE	JO	JS	JNE	JNZ
JNLE	JG	JNB	JAE	JNBE	JA
JNO	JNS				
LOOPE	LOOPNZ	LOOPNE	JCXZ		
IRET					
STC	CLD	STD	CLI	STI	
ESC	LOCK	NOP			

6.1 データ転送命令

8086のデータ転送命令は、汎用データ転送命令、アキュムレータ転送命令、アドレス・オブジェクト転送命令、フラグ・レジスタ転送命令の4種類に分類することができます。

◎ 汎用データ転送命令

MOV
PUSH
POP
XCHG

汎用のデータ転送命令としては、MOV、PUSH、POP、XCHGの4種が用意されています。本書ではすでに紹介した、おなじみのニーモニックばかりでしょう。

MOVは、第1オペランド(デストネーション・オペランド)へ、第2オペランド(ソース・オペランド)のデータを転送(コピー)する働きを持った命令で、オペランドの種類によってバイト・データまたはワード・データを扱うことができます。

次の、**PUSH**および**POP**は、オペランドとスタックの間でデータを転送するための命令で、前者はワード・データをスタックに送り込み、後者は逆にスタックからワード・データを取り出します。なお、PUSHおよびPOPはカレント・スタック・セグメントに対して行なわれるのが常識であるため、たとえば直前にCS:を置いてコードセグメントに対する命令を生成した場合の動作は保証されていません。また、スタックに関しては、3.9節をご参照ください。

最後の**XCHG**は、2つのオペランドで示すバイト・データまたはワード・データをEXCHANGE(交換)するための命令です。

◎ アキュムレータ転送命令

IN
OUT
XLAT

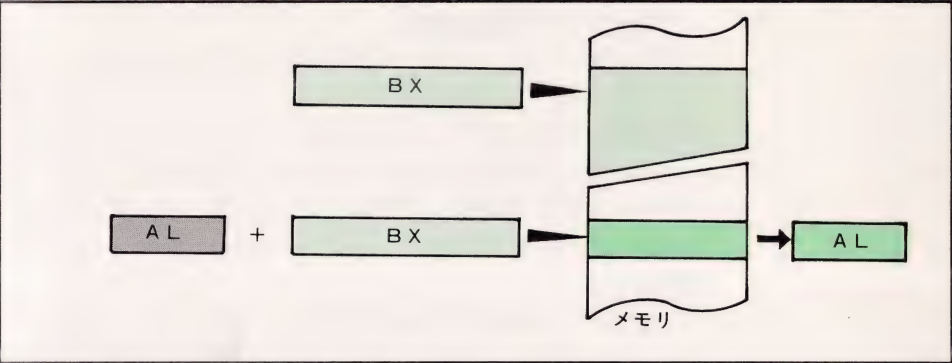
アキュムレータ転送命令はアキュムレータに対してのみ有効な転送命令で、IN、OUT、XLATの3種が用意されています。

INおよび**OUT**は、それらの名称からもわかるように、レジスタALのバイト・データまたはレジスタAXのワード・データを、入力ポートから入力(INPUT)または出力ポートへ出力(OUTPUT)するための命令です。ポートの番号はオペランドとして直接指定するのが一般的ですが、レジスタDXの内容で間接的に指定することもできます。なお、オペランドによって直接指定することのできるポート番号は8ビットのみで、上位8ビットには無条件に0が指定されることに注意してください。

次の**XLAT**は、TRANSLATE(翻訳)の意味を持つ8086ならではの独特な命令です。

具体的なXLATの動作は、レジスタBXの内容にレジスタALの値を加えたものをアドレスとするメモリの内容がレジスタALにロード

されます。したがって、あらかじめレジスタBXにデータ変換テーブルの先頭アドレスを入れておけば、8ビット・データのデータ変換を簡単に行なうことができます。



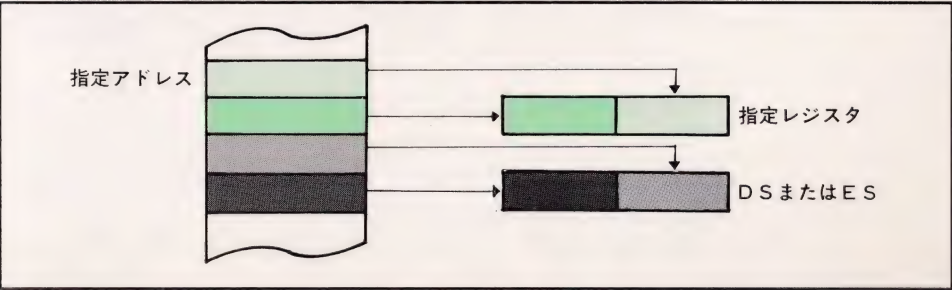
② アドレス・オブジェクト転送命令

8086では、より複雑なアドレス指定（アドレッシング）が可能のように、LEA、LDS、LESの3種のアドレス転送命令が設けられています。

LEA
LDS
LES

LDAは、LOAD EFFECTIVE ADDRESSの意味で、実効アドレスを計算するための命令です。具体的には、第2オペランド（ソース・オペランド）で指定するメモリの具体的なオフセット・アドレスが、第1オペランド（デストネーション・オペランド）で指定する16ビットのレジスタに与えられます。

LDS (LOAD POINTER INTO REGISTER DS)および**LES** (LOAD POINTER INTO REGISTER ES) は、いずれも4バイト（32ビット）のメモリに渡るデータを、2個の16ビット・レジスタに分割して転送するための命令で、おもにセグメント・ベース・アドレスとオフセット・アドレスを対にしたもの（ポインタと言う）を扱うために用意されています。具体的な動作は説明するよりも図示した方がわかりやすいと思い、下図にLDSおよびLESの実行状態を示します。



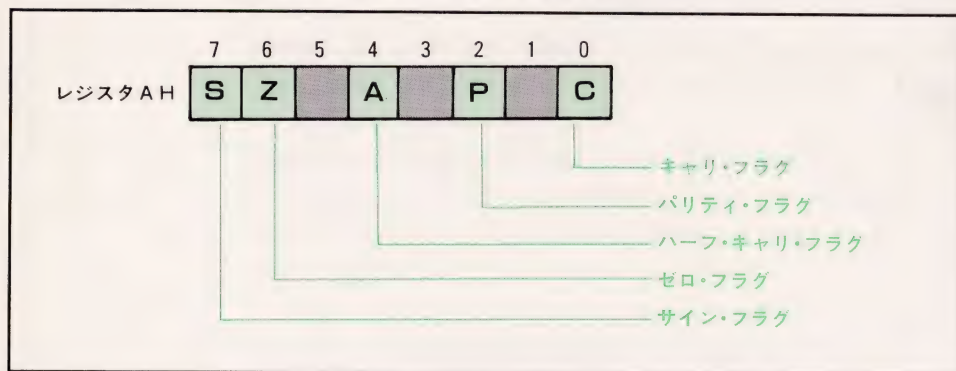
これらの命令、LEA、LDS、LESはともに、ソースは必ずメモリになり、デストネーションは必ず16ビットのレジスタになることにご注意ください。

●フラグ・データ転送命令

LAHF
SAHF
PUSHF
POPF

フラグを、一般のビット・データと同じように扱う場合必要になるのが、このフラグ・データ転送命令で、レジスタAHとフラグの間でデータを転送するLAHF (LOAD REGISTER AH WITH FLAGS)、SAHF (STORE REGISTER AH INTO FLAGS) と、スタックとフラグの間でデータを転送するPUSHF (PUSH FLAGS)、POPF (POP FLAGS) の計4種のニーモニックが存在し、全てオペランドはありません。

次図に、LAHFおよびSAHFの実行時におけるレジスタAHの各ビットとフラグの対応を示します。



この対応を見ていると、レジスタAHに対応しているのはフラグの下位8ビットのみであることに気がつきます。OF、DF、IF、TFの4つのフラグはレジスタAHのどのビットとも対応していません。

これは、LAHFおよびSAHFが、8080用に開発されたプログラムを効率良く実行（移植）するために設けられた命令であり、8086のフラグの下位8ビットは8080のフラグとそのまま対応しているためです。したがって、8080にはOF、DF、IF、TFは存在しません。

PUSHFおよび**POPF**は、8086のフラグ16ビット全てを普通の16ビット・レジスタと同様にして、スタックとの間で転送します。スタックに転送された内容とフラグの内容は3・9節を、フラグに関しては2・7節を、それぞれご参照ください。

6.2 算術演算命令

8086の算術演算命令は、大きく加算系と減算系に分類することができます。また、乗除算も算術演算の一種ですが、本書では項目をかえて説明します。

バイト・データとワード・データの双方を同等に扱い得る点が8086の最大の特徴になっていますが、この算術演算命令も例外ではなく、8ビット、16ビット、無符号、有符号を問わず直接演算を行なうことができます。

算術演算命令を実行することによって、ほとんどのフラグが影響を受けますが、ここで主要なフラグのセットされる条件を表にまとめておきます。表中の条件以外の時、各フラグはクリア（リセット）されます。

F	条件
CF	演算結果が無符号数としてオーバーフローした場合
OF	演算結果が有符号数としてオーバーフローした場合
ZF	演算結果が0（ゼロ）になった場合
SF	演算結果が負（最上位ビットが1）になった場合
PF	演算結果の中で1であるビットの数が偶数である場合
AF	BCDコードの演算結果に対して10進補正が必要な場合

実際に、どの命令を実行することによって、どのフラグが影響を受けるのかを知りたい場合には、巻末付録をご参照ください。

④ 加算命令

加算命令に属する命令は、ADD、ADC（ADD WITH CARRY）、そしてINC（INCREMENT）の3種です。さらに加算用の補正命令として、AAA（ASCII ADJUST FOR ADDITION）およびDAA（DECIMAL ADJUST FOR ADDITION）の2種が用意されています。

加算命令の中で最も一般的なものが、**ADD**です。ADDは第1オペランドが示すデストネーションに、第2オペランドで示すソースを加えるための命令で、片方の被演算項はレジスタまたはメモリで、もう一方はレジスタかイミディエント・データ（直接データを与えることができる）となっています。

ADCは、キャリ・フラグ（CF）の値（0または1）を同時に加える点を除けば、ADDと変わらない機能を持ち、ADDを1度実行しただけでは不可能な多桁にわたる加算（16ビットを越える加算）を行なう場合に有用です。

さらに、レジスタまたはメモリの内容に1を加えることは多いため、

ADD
ADC
INC
AAA
DAA

BCDコード

1を加える専用の単項演算命令 **INC** が用意されています。

AAA および **DAA** は、加算における、それぞれ、アンパック形式のBCDコード、パック形式のBCDコードのための補正命令です。
AAAを実行すると、ハーフ・キャリ・フラグ (AF) が1またはレジスタALの下位4ビットが0AH以上の場合にのみ、レジスタAXに0106Hが加えられ、ハーフ・キャリ・フラグ (AF) とキャリ・フラグ (CF) がセットされた上にレジスタALの上位4ビットがクリアされます。

DAAの動作は、8080 (Z80) のものと同様に、ハーフ・キャリ・フラグ (AF) が1またはレジスタALの下位4ビットが0AH以上の場合にのみ、レジスタALに06Hを加え、ハーフ・キャリ・フラグ (AF) をセットします。その結果キャリ・フラグ (CF) が1またはレジスタALがA0H以上の場合には、さらにレジスタALに60Hを加え、キャリ・フラグ (CF) をセットします。

◎減算命令

SUB

SBB

DEC

NEG

CMP

AAS

DAS

減算に属する命令は、SUB (SUBTRACT)、SBB (SUBTRACT WITH BORROW)、DEC (DECREMENT)、NEG (NEGATE)、CMP (COMPARE) の5種、そして、補正命令AAS(ASCII ADJUST FOR SUBTRACTION)、DAS (DECIMAL ADJUST FOR SUBTRACTION) の計7種です。

SUB および **SBB** は、加算命令のADDおよびADCに相当するもので、SUBは一般的な減算を行なう命令であり、SBBはさらにキャリ・フラグ (CF) の内容 (0または1) を含めて減算を行ないます。

DEC は、オペランドが示す内容から1を減じるための命令で、オペランドにはレジスタまたはメモリが許されています。

NEG は、オペランドが示すデータを有符号と見なして、その符号を反転 (0から減じる) します。

次の**CMP**は、条件判断の時に多用される比較命令で、演算自体はSUBと全く同じ減算を行ないませんが、減算結果 (差) を求めることはできず、各種のフラグのみが影響を受けます。

演算結果を求める事のできない演算命令を使うことにとまどいを感じる皆さんもおられるかも知れませんが、CMPのように2つのデータの間で大小関係のみを知りたい状況はよくあるものです。

8086には、SUBに対応する命令としてこのCMPが用意されているように、論理積を求める命令ANDに対応した**TEST**も用意されており、やはりフラグのみに影響を与えます。

次ページの表は、第1オペランドとして与えるデストネーション・データをXとし、第2オペランドとして与えるソース・データをYとして

CMPを実行した場合の、XYの関係および主なフラグの変化です。

データ	条件	CF	OF	SF	ZF
有符号／無符号	X=Y (等)	0	0	0	1
有符号	X<Y (小)	U	0	1	0
	X<Y (小)	U	1	0	0
	X>Y (大)	U	0	0	0
	X>Y (大)	U	1	1	0
無符号	X<Y (下)	1	U	U	0
	X>Y (上)	0	U	U	0

残りの命令、**AAS**および**DAS**は、それぞれ減算時の補正命令で、AASはアンパック形式のBCDコードのために、DASはパック形式のBCDコードのために補正を行いません。

AASは、ハーフ・キャリ・フラグ (AF) が1かまたはレジスタALの下位4ビットが0AH以上の場合にのみレジスタALから06Hを減じ、ハーフ・キャリ・フラグ (AF) とキャリ・フラグ (CF) をセットした上にレジスタAHから01Hを減じ、レジスタALの上位4ビットをクリアします。

DASでは、ハーフ・キャリ・フラグ (AF) 1かまたはレジスタALの下位4ビットが0AH以上の場合にのみレジスタALから06Hを減じ、ハーフ・キャリ・フラグ (AF) をセットします。その結果キャリ・フラグ (CF) が1かまたはレジスタALがA0H以上の場合にはさらにレジスタALから60Hを減じ、キャリ・フラグ (CF) をセットします。

補正命令AAA、DAA、AAS、DASは、BCDコード (パック形式またはアンパック形式) のデータ間で各種の演算を実行した場合の補正を行なうために用意された命令ですから、算術演算の直後に置いて実行し、補正命令のみが単独で実行されることはまずありません。

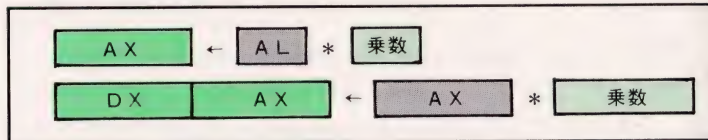
6.3 乗除算命令

8086では、無符号データおよび有符号データの乗除算を行なうことができます。さらにBCDコードの乗除算を行なうために、アンパック形式のBCDコードとバイナリ・コード（2進数）の間で形式を変換する命令、そして8ビット長のデータを16ビット長に、16ビット長のデータを32ビット長に拡張する命令等も存在し、さまざまな形式のデータを扱いやすいように考慮されています。

●乗算命令

乗算命令には、無符号データのための**MUL** (MULTIPLY) および有符号データのための **IMUL** (INTEGER MULTIPLY) が用意されており、それぞれがオペランドのビット長によって8ビット乗算と16ビット乗算に分類できます。

乗数としてはレジスタかメモリ上のデータをオペランドによって指定することができますが、被乗数は8ビット乗算の場合レジスタALに、16ビット乗算の場合レジスタAXに与えなければならない、演算結果（積）はレジスタAXまたはレジスタDX、レジスタAXに渡って求められます。



上記の**MUL**および**IMUL**の他に乗算命令に含まれるものとして**AAM** (ASCII ADJUST FOR MULTIPLY) があげられます。**AAM** はアンパック形式のBCDコード間で乗算命令を実行した後の補正を行なうために用意されていますが、**AAM**が実際に行なうことは2進数（バイナリ・コード）をBCDコードに変換することに他なりません。

すなわち、**AAM**を実行することで、レジスタALの2進データ(0～99)を10 (0AH) で割った商がレジスタAHに、剰余がレジスタALに格納されると考えるのがわかりやすいと思います。

なお、この命令は、除算命令の実行結果をアンパック形式のBCDコードに変換する目的でも使用します。

●除算命令

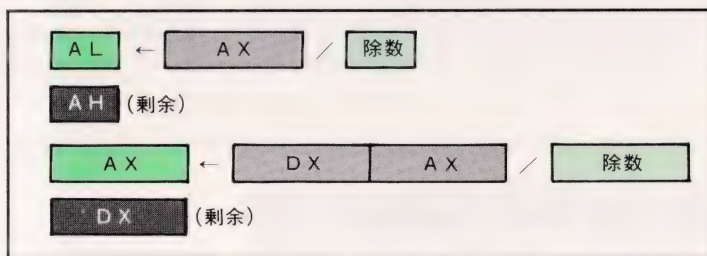
除算命令も乗算の場合と同じ様に8ビットと16ビットのデータを扱う命令が存在し、無符号データのための**DIV** (DIVIDE) および有符号データのための **IDIV** (INTEGER DIVIDE) が用意されています。

除数はレジスタかまたはメモリ上のデータをオペランドによって指

MUL
IMUL
AAM

DIV
IDIV
AAD

定することができますが、被除数は8ビット除算の場合レジスタAXに、16ビット除算の場合レジスタDX、レジスタAXに渡って与えなければならない、商はレジスタALまたはレジスタAXに、剰余はレジスタAHまたはレジスタDXに求められます。



除算命令の実行時に、商がオーバーフローを起こした場合、たとえば除数が0（ゼロ）の場合や8ビットの除算命令で $512/2$ を実行した場合には、タイプ0の割り込みが発生しますが、N88-BASIC(86)では割り込み処理ルーチンから何もせずにもどります。

また、IDIVによって有符号データの除算を行なう場合には符号の取り方によって商と剰余の組み合わせが2通り考えられることもありますが、そのような場合に8086は剰余の符号が被除数の符号と一致する解の組み合わせを選択します。

次に、除算用の補正命令AADですが、この命令の使用方法は他の補正命令と異なっています。なぜならば、加減乗算の場合にはBCDコードを通常のバイナリ・コードと見なして強引に演算を行なってしまう、直後の補正によって完全なBCDコードに直すことが可能でしたが、除算の場合には補正が行なえるとは限らないからです。

したがって、アンパック形式のBCDコードは、一度バイナリ・コードに変換してから除算命令を実行することになりますが、このBCDコードからバイナリ・コードへの変換を行なう命令が、他ならないAADです。当然、演算結果はバイナリ・コードで求められますので、もし演算結果がBCDコードで欲しい場合には、AAMによってアンパック形式のBCDコードに変換し直してやらなければなりません。

AADは、レジスタAHのデータを10倍してレジスタALに加える命令と言い換えると考えやすいと思います。

乗除算命令の残りは、CBW (CONVERT BYTE TO WORD) およびCWD (CONVERT WORD TO DOUBLE WORD) ですが、それぞれ有符号のバイト・データをワード長に、ワード・データをダブルワード長に拡張する命令です。

CBW

CWD

6.4 論理演算命令

本節では、8086の論理演算命令を、シングル・オペランド論理命令（単項演算系）とダブル・オペランド論理命令（2項演算系）に分類して説明します。実は、この分類方法は一般的ではなく、本来ならば、ローテート（回転）シフト命令のための別の見出しを設けたいところなのですが、巻末付録の分類基準にそるえるために、オペランドの数による分類方法を採用しました。

④ シングル・オペランド論理命令

シングル・オペランド論理命令の中では、**NOT**のみが純粋な論理演算（ブール演算）と言って良く、他の命令は全てローテート・シフト命令と呼ばれているものです。

次表に、8種のローテート・シフト命令の意味を示します。

命令	意味
SHL	SHIFT LOGICAL LEFT
SAL	SHIFT ARITHMETIC LEFT
SHR	SHIFT LOGICAL RIGHT
SAR	SHIFT ARITHMETIC RIGHT
ROL	ROTATE LEFT
ROR	ROTATE RIGHT
RCL	ROTATE THROUGH CARRY LEFT
RCR	ROTATE THROUGH CARRY RIGHT

SHLと**SAL**は、ニーモニックは2種類存在しますが、動作内容が全く同じであるため、オブジェクト・コードは同一のものが使用されます。

次ページに、各ローテート・シフト命令の動作内容を示します。

なお、ローテート・シフト命令の存在理由や具体的な使用例については、5.7節で詳しく説明しましたので、ここでは省略させていただきます。

④ ダブル・オペランド論理命令

2つのデータ間で論理演算を行なう命令には、**AND**、**TEST**、**OR**、**XOR**（EXCLUSIVE-OR）の4種類が存在します。

この中の**TEST**は、**CMP**と同様に演算結果がフラグのみに残される命令で、実際には**AND**（論理積）を求めますが、論理積自体を得ることはできません。この**TEST**は、バイトまたはワード・データ中の特定のビットが0であるか1であるかをチェックする場合などに多用されます。

NOT
SHL
SAL
SHR
SAR
ROL
ROR
RCL
RCR

AND
TEST
OR
XOR

ローテートおよびシフト命令の動作

命令	動作
SHL SAL	
SHR	
SAR	
ROL	
ROR	
RCL	
RCR	

6.5 スtring操作命令

8086には、メモリ上に置かれた一連のデータ・ブロック（String）を操作するための命令が数多く用意されており、これらは通常の場合**リピート・プリフィクス命令**と**プリミティブ・String命令**が対で使用されます。

全てのString命令では、データ・ブロックの位置するデストネーション・アドレスを指定するために、エクストラ・セグメント・レジスタ（ES）およびデストネーション・インデックス・レジスタ（DI）を用い、1バイト（1ワード）のデータ処理後、レジスタDIの値が自動的に1（または2）ずつ増減されます。

また、2組のデータ・ブロックを同時に扱うString命令では、データ・セグメント・レジスタ（DS）とソース・インデックス・レジスタ（SI）を用いて、ソース・アドレスを指定しますが、レジスタSIの値も自動的に1（または2）ずつ増減されます。

さらに、データ・ブロックの長さを指定することが必要ですが、これはレジスタCXに与え、命令の実行はレジスタCXをカウンタとしてディクリメントして行き、レジスタCXが0（ゼロ）になるまでくり返されます。

上記の他に、ディレクション・フラグ（DF）の内容が、データ・ブロックに対する処理の方向を決定します。すなわち、これが0の場合にはレジスタDIおよびレジスタSIの値が増されて行き（順方向）、これが1の場合には減らされて行きます（逆方向）。

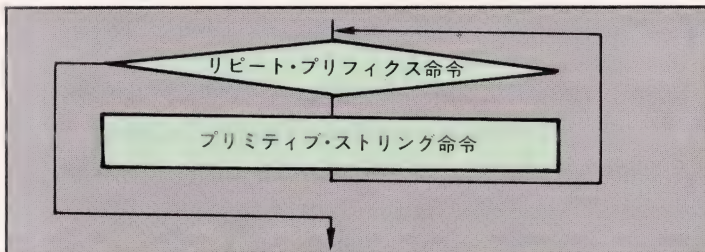
String操作命令の実行にあたって、各レジスタに与えるべき内容をまとめておきます。

ES	← デストネーションのセグメント・アドレス
DI	← デストネーションのオフセット・アドレス
DS	← ソースのセグメント・アドレス
SI	← ソースのオフセット・アドレス
CX	← データ・ブロックのバイト（ワード）長
DF	← 処理の方向（順方向の場合0）

◎リピート・プリフィクス命令

リピート・プリフィクス命令としては、REP、REPE、REPZ、REPNE、REPNZの5種のニーモニックが用意されていますが、REPEとREPZ、REPNEとREPNZは、それぞれ全く同じコードですから、実際には3種の命令が存在することになります。

リピート・プリフィクス命令は、直後に置かれたプリミティブ・ストリング命令をくり返す条件を指定するためのプリフィクス命令で、両者を組み合わせることによって、次のフローチャートで表現する処理を構成することができます。



最も基本的なリピート・プリフィクス命令は、**REP**で、レジスタCXの値が0（ゼロ）になるまで、プリミティブ・ストリング命令をくり返します。一般的には、MOVS、MOVSB、MOVSW、STOSなどと組み合わせて使用します。

次の**REPE**および**REPZ**は、レジスタCXの値が0（ゼロ）にならなくても、ゼロ・フラグ（ZF）がセットされていれば処理を中断し、CMPS、SCASなどと組み合わせて使用します。

REPNEおよび**REPNZ**は、逆にゼロ・フラグ（ZF）が0の場合にも処理を中断します。

なお、ストリング処理中断の判定は、各レジスタ（DI、SI、CX）の値が更新させた後に行なわれます。またレジスタCXの値が0（ゼロ）の時にストリング操作命令を実行した場合に、8086は何もせずに次の命令に進みます。

◎プリミティブ・ストリング命令

プリミティブ・ストリング命令は、一般的に使用される状況によって、さらに2種類に分けることができます。

一方は、リピート・プリフィクス命令とともに組み合わせるだけで、ある程度の機能を発揮することができるもので、MOVS、CMPS、SCASが属します。

MOVSは、データの転送を行なうための命令であり、REPと組み合わせることによって、バイト・データ群またはワード・データ群の転送を簡単に実行することができます。もちろん、あらかじめレジスタ

REP
REPE
REPZ
REPNE
REPNZ

MOVS
MOVSB
MOVSW
CMPS
SCAS
LODS
STOS

ESおよびレジスタDIによって転送先のアドレスが、レジスタDSおよびレジスタSIによってデータ群の格納されているアドレスが指定されており、さらにレジスタCXに転送するデータの個数を与えておかなければなりません。

次の**CMPS**は、ソース・インデクス(この場合レジスタDSおよびレジスタSI)によって指定されるメモリの内容と、デストネーション・インデクス(レジスタESおよびレジスタDI)によって指定されるメモリの内容と比較する機能を持っており、REPE (REPZ)またはREPNE (REPNZ)と組み合わせることで、データ・ブロックの一致、不一致を検出する命令を構成できます。なお、CMPSの実行によるフラグへの影響は、一般の比較命令 (CMP) と同等です。

同様に、REPE (REPZ) またはREPNE (REPNZ) と組み合わせることによって、データ・ブロック中から一定のデータをさがし出すために使用できるのが**SCAS**です。この命令SCASを実行すると、レジスタALのバイト・データまたはレジスタAXのワード・データと、デストネーション・インデクスによって指定するメモリのバイト・データまたはワード・データを比較して、結果をフラグにセットします。

8086には、さらにリピート・プリフィクス命令を伴わずに使用する2種の命令、LODSとSTOSが用意されています。

LODSおよび**STOS**は、ストリング操作命令としての基本的な動作(レジスタDI、SI、CXの操作)を行ないますが、リピートプリフィクス以外の多くの命令と組み合わせて、より複雑なストリング操作を行なうために設けられています。したがって処理の反復は、普通の分岐命令によることがほとんどです。具体的な動作は、LODSはソース・インデクスからアキュムレータへのロード、STOSはアキュムレータからデストネーション・インデクスへのストアです。

プリミティブ・ストリング命令全般にわたって注意すべき点は、次表の様にダミーのオペランドを付加し、扱うデータがバイト長であるかワード長であるかを明確にしなければならないことです。

バイト長		ワード長	
MOVS	AL,AL	MOVS	AX,AX
MOVSB		MOVSW	
CMPS	AL,AL	CMPS	AX,AX
SCAS	AL	SCAS	AX
LODS	AL	LODS	AX
STOS	AL	STOS	AX

6.6 分岐命令

一般的なジャンプ命令、サブルーチン・コール命令、リターン命令から、割り込み処理ルーチンへの分岐命令まで、プログラム実行の流れを変更する全ての命令を、この分岐命令と見なすことができます。

分岐命令はさらに、(無条件)制御転送命令、条件付制御転送命令、くり返し転送命令、割り込み転送命令、の4種に分類できます。

◎制御転送命令

一般的には、無条件分岐命令などと呼ばれている、CALL、JMP、RETが制御転送命令に属します。

JMPは、命令ポインタ (IP) と場合によってはコード・セグメントレジスタ (CS) に新しい値を代入することによって、実行の流れを変更する働きを持ちます。**CALL**も同様にして実行の流れを変える命令ですが、最初に現在の命令ポインタ (IP) と場合によってはコード・セグメント・レジスタ (CS) の値をスタックに待避しておく点異なります。

RETは、CALLの実行によってスタックに待避されていた値を命令ポインタ (IP) などにもどすことで、サブルーチンがコールされた以前の流れにもどる働きを持ちます。

これらの命令は、分岐する先のアドレスがカレント・コード・セグメントの中であるかどうかによって、セグメント内分岐用とセグメント間分岐用の2種類が用意されており、両者の区別方法はアセンブラによって異なりますが、後者のセグメント間分岐の場合にはニーモニックの末尾にFARのFを付加し、**CALLF**、**JMPF**、**RETF**などと表現するのが一般的です。

セグメント内、セグメント間を問わず、JMPおよびCALLを実行する場合には分岐先のアドレスを与えることが必要であり、セグメント内であればオフセット・アドレスのみを与えますが、セグメント間の場合にはコード・セグメント・レジスタ (CS) に設定するためのベース・アドレスもあわせて指定しなくてはなりません。この分岐先アドレスは各命令のオペランドとして指定しますが、8086では直接アドレスを与える他に、レジスタの内容やメモリ上の (ダブル) ワード・データさえも分岐先アドレスとして指定することができます。

さらに、8086の特色として、リターン命令 (RET) にもオペランドが存在する点をあげることができます。もちろん、RETのオペランドは分岐先のアドレスではありませんし、必要ないことがほとんどです。オペランド付きのRETでは、サブルーチンからもどる際に、ス

CALL
JMP
RET

CALLF
JMPF
RETF

タック・ポインタ (SP) にオペランド (0~65536) が加えられます。これは、メイン・ルーチンからサブルーチンへの引数をスタックを介して渡すような場合に、不用になった引数領域を簡単に廃棄するために設けられた機能です。

●条件付制御転送命令

条件分岐命令については4・1節で詳述してありますので、ここでは簡単な表だけをあげておきます。ただし、これらの命令は現在の命令ポインタ (IP) から-127~+128の範囲でしか、分岐を行なうことができますので、JMPと組み合わせなければならない場合もあります。

命令	条件	条件
JE	ZF=1	EQUAL
JZ		ZERO
JL	(SF≠OF)=1	LESS
JNGE		NOT GREATER NOR EQUAL
JLE	((SF≠OF)∨ZF)=1	LESS OR EQUAL
JNG		NOT GREATER
JB	CF=1	BELOW
JNAE		NOT ABOVE NOR EQUAL
JBE	(CF∨ZF)=1	BELOW OR EQUAL
JNA		NOT ABOVE
JP	PF=1	PARITY
JPE		PARITY EVEN
JO	OF=1	OVERFLOW
JS	SF=1	SIGN
JNE	ZF=0	NOT EQUAL
JNZ		NOT ZERO
JNL	(SF≠OF)=0	NOT LESS
JGE		GREATER OR EQUAL
JNLE	((SF≠OF)∨ZF)=0	NOT LESS NOR EQUAL
JG		GREATER
JNB	CF=0	NOT BELOW
JAE		ABOVE OR EQUAL
JNBE	(CF∨ZF)=0	NOT BELOW NOR EQUAL
JA		ABOVE
JNP	PF=0	NOT PARITY
JPO		PARITY ODD
JNO	OF=0	NOT OVERFLOW
JNS	SF=0	NOT SIGN

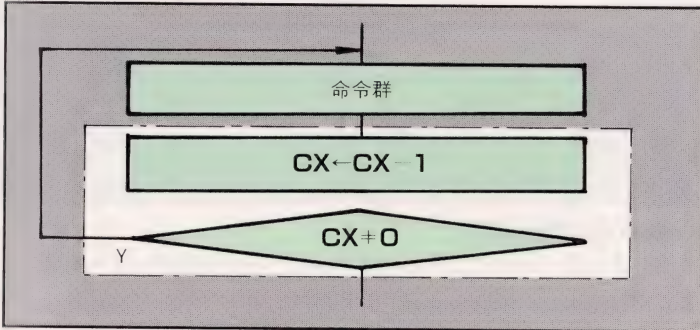
③くり返し転送命令

くり返し（反復）転送命令は、レジスタCXをカウンタとして、一定範囲内の命令群をくり返し実行するための命令です。なお、これらくり返し転送命令によって分岐できる範囲も、先の条件付制御転送命令と同様に、命令ポインタの値から-128～+127のアドレスのみです。

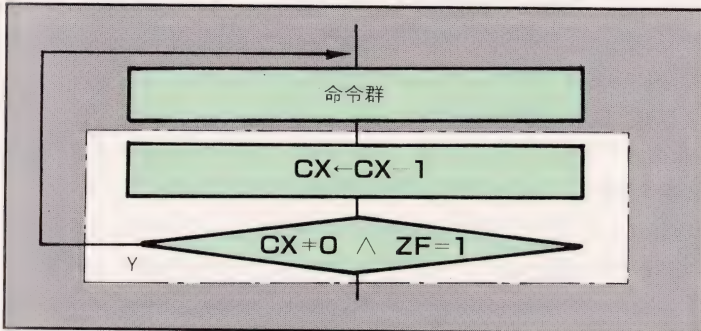
くり返し転送命令には、LOOP、LOOPZ、LOOPE、LOOPNZ、LOOPNE、JCXZの6種のニーモックが設けられています。ただし、LOOPEとLOOPZ、LOOPNEとLOOPNZは、それぞれ同じコードとなりますので、実際に用意されているのは4種の命令のみです。

くり返し転送命令の基本は**LOOP**です。この命令を実行すると、レジスタCXの値がディクリメントされ、さらにレジスタCXの値が0（ゼロ）でなければ、オペランドで指定するアドレスに分岐します。

この様子をフローチャートに表現して示します。点線で囲った2つの箱がLOOPの持つ機能です。



同様にして、**LOOPZ (LOOPE)** の一般的な使用方法も、フローチャートにしました。LOOPZは、レジスタCXの値をディクリメントした後、レジスタCXの値が0（ゼロ）でないだけでなく、さらにゼロ・フラグ（ZF）がセットされている場合に分岐します。

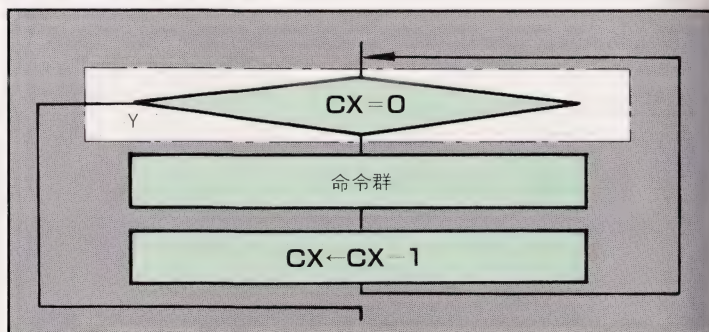


LOOP
LOOPZ
LOOPE
LOOPNZ
LOOPNE
JCXZ

ここで注意しなければならないのは、くり返し転送命令の実行によるレジスタCXのデクリメントが、ゼロ・フラグ (ZF) に全く影響を与えないことです。したがって、くり返し転送命令によるゼロ・フラグ (ZF) の判断は、事前の命令群に対して行なわれることになります。

LOOPNZ (LOOPNE) は、分岐条件が、レジスタCXが0 (ゼロ) でなく、かつゼロ・フラグ (ZF) が0である点を除けば、先のLO OPZと同じです。

最後の**JCXZ**は、レジスタCXの値が0の場合にのみ分岐を行なう命令で、頭にLOOPが付く他の命令とは逆に、ループ中から脱出するために使用すると便利です。



④ 割り込み転送命令

INT は、ハードウェア的な割り込みによらず割り込み処理ルーチン呼び出すための命令です。割り込みのタイプ (0~255) は、オペランドで指定しますが、タイプを3に指定した場合にのみ、1バイトの特別なコードが用意されます。これは、タイプ3の割り込みがブレーク・ポイントを設定するためのものであるため、1バイト命令の方が他の命令と置き換えやすいからです。

さらに、割り込みを発生するための命令として、**INTO**があります。これは、オーバフロー・フラグ (OF) が1の場合にのみ、タイプ4の割り込みを発生することができます。

なお、以上の命令がサブルーチン・コール命令と異なる点は、コード・セグメント・レジスタ (CS) と命令ポインタ (IP) の値をスタックに退避する前に、全てのフラグ (F) を退避することです。

逆に、**IRET**は割り込み処理ルーチンからもどるための命令で、スタックに退避されているもどり先のアドレスを命令ポインタ (IP) とコード・セグメント・レジスタ (CS) に与えた後、フラグ (F) の値も元通りに設定し直します。

INT
INTO
IRET

6・7 CPU制御命令

8086のCPU制御命令は、いくつかのフラグを単独に操作することのできるフラグ制御命令と、その他のCPU制御命令に分類されています。

④フラグ制御命令

フラグ制御命令を実行することによって、キャリ・フラグ (CF)、ディレクション・フラグ (DF)、インタラプト・イネーブル・フラグ (IF) のそれぞれを、独立してセットまたはクリアすることができます。

ニーモニックは、**CLC** (CLEAR CARRY)、**STC** (SET CARRY)、**CLD** (CLEAR DIRECTION)、**STD** (SET DIRECTION)、**CLI** (CLEAR INTERRUPT)、**STI** (SET INTERRUPT) となっています。

また、キャリ・フラグ (CF) に関しては、それを反転する命令、**CMC** (COMPLEMENT CARRY) も用意されています。

④CPU制御命令

CPU制御に属する命令は、**HLT** (HALT)、**WAIT** (WAIT)、**ESC** (ESCAPE TO EXTERNAL DEVICE)、**LOCK** (BUS LOCK PREFIX)、**NOP** (NO OPERATION) の5種です。

HLTは、本書掲載のほとんどのサンプル・プログラムが最後で使っている命令で、この命令を実行すると8086はホールド状態に入ります。ホールド状態からぬけ出すためには外部割り込みを発生させるしかありません。8086は割り込み処理を終えると、HLTの次の命令から実行を再開します。

本章でも紹介して来たように、8086は非常に多くの命令セットを持っています。しかし、8086を搭載したパーソナル・コンピュータを、より汎用的に使うと考えた場合には、さらに多くの命令を欲しくなって当然と言えましょう。システムがますます複雑になってきている現在では、全ての機能を8086のみに期待することはできないのです。

そこで、8086に欠けた機能を下位のプロセッサに任せることになります。8086の下位プロセッサとして最も一般的なのが、浮動小数点の演算機能を受け持つ8087です。8086からはエスケープ命令 (ESC) で、浮動小数点演算を呼び出しますが、8087を付加してある場合、8086がESCをフェッチするたびに、8087がかわりに命令を実行します。これに関して、プログラマは8086と8087のどちらが命令を実行しているのかを意識する必要がない

CLC
CMC
STC
CLD
STD
CLI
STI

HLT
WAIT
ESC
LOCK
NOP

8087

ため、あたかも8086の命令が拡張された感覚でプログラミングを行なうことができます。

すなわち、8086と8087は同じプログラムを常に平行して実行して行き、8086の命令であれば8086が実行し、ESCであれば8087が実行するわけです。ESCには2つのオペランドが存在し、第1オペランド(0~63)で下位プロセッサの種類および命令の形式を64通りに指定し、第2オペランドでレジスタまたはメモリを指定することができます。

次の**WAIT**は、下位プロセッサとの同期をとるための命令で、この命令が実行されると8086は、下位プロセッサが演算を終えるまで動作を停止します。演算結果が得られるのを待つことができるのはもちろん、ESCの直前にWAITを置くことによって下位プロセッサの作業が終了する前に次の命令を送ってしまうミスを防ぐことができます。

もし、ウェイト状態の時に外部割り込みの要求があった場合には、割り込みを発生し、復帰後再びウェイト状態に入ります。

LOCKは、マルチ・プロセッサのためのプリフィクス命令であり、このロック・プリフィクス直後の命令を実行する間、他のプロセッサがメモリのアクセスを行なわないように、アドレス・バスをロックする意味で設けられています。これは、セマフォ変数をチェックする場合などのように、同時に複数のプロセッサが同じアドレスをアクセスしてはならない場合を考慮したものです。

最後の**NOP**は説明の必要もないと思います。8086にはNOPのコードとして00Hではなく90Hが割り当てられていますが、これはレジスタAXとレジスタAXの間で内容を交換する命令、

XCHG AX, AX

のコードと同じものです。

7章 [付録] μ PD8086
インストラクション活用表

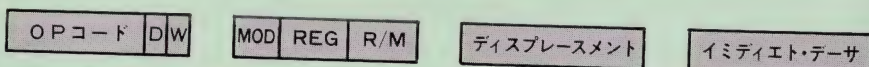
7.0 8086インストラクション活用表

本章では、8086の全インストラクションについて、ニーモニック、オペランド、オペレーション・コード(機械語)、命令のバイト数、実行クロック数、オペレーション内容、フラグの変化状況を、表にまとめて示します。

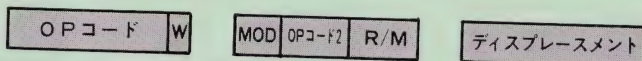
Z80等の活用表は販売されていますが、8086のものはほとんど見かけませんので、お役に立つと思います。

●命令の基本形

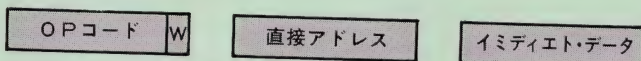
オペランドとしてレジスタとレジスタ/メモリをもつ命令



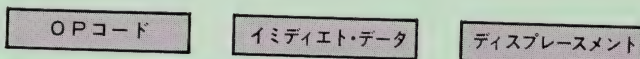
オペランドとしてレジスタ/メモリをもつ命令



アキュムレータを対象とする命令



その他の命令



●命令語の汎例

識別子	意味
W	ワード/バイト・フィールド (0~1)
reg	レジスタ・フィールド (000~111)
r/m	レジスタ/メモリ・フィールド (000~111)
mod	モード・フィールド (00~10)
S:W	S:W=01のときdata=16ビットでそれ以外はdata=8ビット S:W=11のときバイト・データのサインが拡張されワード・オペランド
XXX	ESCオペコードの1番目の3ビット
YYY	ESCオペコードの2番目の3ビット

命令機能説明の汎例

識別子	意味
AX	レジスタ A X
AH	レジスタ A H
AL	レジスタ A L
BX	レジスタ B X
CX	レジスタ C X
CL	レジスタ C L
DX	レジスタ D X
SP	スタック・ポインタ
IP	命令ポインタ
FLAGS	フラグ
DI	デストネーション・インデクス・レジスタ
SI	ソース・インデクス・レジスタ
CS	コード・セグメント・レジスタ
ES	エクストラ・セグメント・レジスタ
AF	ハーフ・キャリ・フラグ
CF	キャリ・フラグ
PF	パリティ・フラグ
SF	サイン・フラグ
ZF	ゼロ・フラグ
DF	ディレクション・フラグ
IF	インタラプト・イネーブル・フラグ
OF	オーバフロー・フラグ
TF	トラップ・フラグ
(.....)	() 内で示すレジスタまたはメモリ
((.....))	(()) 内で間接的に指定したメモリ
:	ワードにわたるアドレス
disp	ディスプレイースメント
temp	テンポラリ・レジスタ
tmpcf	テンポラリ・キャリ・フラグ
seg	セグメント・アドレス
offset	オフセット・アドレス
EXT	拡張バイトまたはワード
LOW	下位バイトまたはワード
←	転送方向
+	加算
-	減算

*	乗算
/	除算
%	剰余
&	論理積
∨	論理和
⊖	排他的論理和
××××H	16進数

◎オペランド・タイプの汎例

識別子	意味
reg	8/16ビット汎用レジスタ
reg8	8ビット汎用レジスタ
reg16	16ビット汎用レジスタ
mem	8/16ビット・メモリ・ロケーション
mem8	8ビット・メモリ・ロケーション
mem16	16ビット・メモリ・ロケーション
mem32	32ビット・メモリ・ロケーション
imm	16ビット・イミディエト・データ
imm8	8ビット・イミディエト・データ
acc	レジスタ A X またはレジスタ A L
sreg	セグメント・レジスタ
source-table	256バイト翻訳テーブルの名称
dst-string	デストネーション・ストリングの名称
src-string	ソース・ストリングの名称
near-proc	カレント・コード・セグメント内のプロシージャ (サブルーチン)
far-proc	カレント・コード・セグメント外のプロシージャ (サブルーチン)
near-label	カレント・コード・セグメント内のラベル
short-label	命令ポインタから-128~+127の範囲のラベル
far-label	カレント・コード・セグメント外のラベル
memptr16	制御が移されようとしているカレント・コード・セグメント内のロケーションのオフセットを含むワード
memptr32	制御が移されようとしている別のコード・セグメント内のロケーションのオフセットとセグメント・ベース・アドレスを含むダブル・ワード
regptr16	制御が移されようとしているカレント・コード・セグメント内のロケーションのオフセットを含む16ビット汎用レジスタ
repeat	ストリング命令のリピート・プリフィクス
pop-value	スタックから捨てるバイト数で通常は偶数
ext-op	コプロセッサの命令中にエンコードされる0~63のイミディエト・データ

●フラグ動作の汎例

識別子	意味
(ブランク)	変化なし
0	0にクリアされる
1	1にセットされる
X	結果にしたがってセットまたはクリアされる
U	不定
R	以前に退避した値がリストアされる

●メモリ・アドレッシング

r/m	mod	00	01	10
000		(BX) + (SI)	(BX) + (SI) + disp8	(BX) + (SI) + disp16
001		(BX) + (DI)	(BX) + (DI) + disp8	(BX) + (DI) + disp16
010		(BP) + (SI)	(BP) + (SI) + disp8	(BP) + (SI) + disp16
011		(BP) + (DI)	(BP) + (DI) + disp8	(BP) + (DI) + disp16
100		(SI)	(SI) + disp8	(SI) + disp16
101		(DI)	(DI) + disp8	(DI) + disp16
110		disp16	(BP) + disp8	(BP) + disp16
111		(BX)	(BX) + disp8	(BX) + disp16

●レジスタの選択

reg or r/m	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

sreg	選択
00	ES
01	CS
10	SS
11	DS

●アドレッシング・モードと実行クロック数

アドレッシング・モード	E A
16ビットのオフセット・アドレスによる直接指定	6
ベースまたはインデックス・レジスタによる間接指定	5
ディスプレースメントを伴うベースまたはインデックス・レジスタによる間接指定	9
ベースとインデックス・レジスタの和による間接指定	7/8
ディスプレースメントを伴うベースとインデックス・レジスタの和による間接指定	11/12

奇数アドレスから始まるワード・データの指定にはさらに4クロック
 加え、セグメント・オーバーライドには2クロックが必要です。

7.1 データ転送命令

◎ 汎用データ転送命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
MOV	reg, reg	1 0 0 0 1 0 0 W	1 1 reg r/m	2	2	
	mem, reg	1 0 0 0 1 0 0 W	mod reg r/m	2-4	9+E A	
	reg, mem	1 0 0 0 1 0 1 W	mod reg r/m	2-4	8+E A	
	mem, imm	1 1 0 0 0 1 1 W	mod 0 0 0 r/m	3-6	10+E A	
	reg, imm	1 0 1 1 W reg		2-3	4	
	acc, mem	1 0 1 0 0 0 0 W		3	10	
	mem, acc	1 0 1 0 0 0 1 W		3	10	
	sreg, reg16	1 0 0 0 1 1 1 0	1 1 0 sreg r/m	2	2	
	sreg, mem	1 0 0 0 1 1 1 0	mod 0 sreg r/m	2-4	8+E A	
	reg16, sreg	1 0 0 0 1 1 0 0	1 1 0 sreg r/m	2	2	
	mem, sreg	1 0 0 0 1 1 0 0	mod 0 sreg r/m	2-4	9+E A	
PUSH	mem	1 1 1 1 1 1 1 1	mod 1 1 0 r/m	2-4	16+E A	
	reg	0 1 0 1 0 reg		1	10	
	sreg	0 0 0 sreg 1 1 0		1	10	
POP	mem	1 0 0 0 1 1 1 1	mod 0 0 0 r/m	2-4	17+E A	
	reg	0 1 0 1 1 reg		1	8	
	sreg	0 0 0 sreg 1 1 1		1	8	
XCHG	reg, reg	1 0 0 0 0 1 1 W	1 1 reg r/m	2	4	
	mem, reg	1 0 0 0 0 1 1 W	mod reg r/m	2-4	17+E A	
	AX, reg16	1 0 0 1 0 reg		1	3	

	オペレーション	フラグ					
		A	C	O	P	S	Z
	(reg) ← (reg)						
	(mem) ← (reg)						
	(reg) ← (mem)						
	(mem) ← (data)						
	(reg) ← (data)						
	if W=0 (AL) ← (addr)						
	if W=1 (AX) ← (addr+1 : addr)						
	if W=0 (addr) ← (AL)						
	if W=1 (addr+1 : addr) ← (AX)						
	(sreg) ← (reg16)						
	(sreg) ← (mem)						
	(reg16) ← (sreg)						
	(mem) ← (sreg)						
	(SP) ← (SP) - 2						
	((SP) + 1 : (SP)) ← (mem)						
	(SP) ← (SP) - 2						
	((SP) + 1 : (SP)) ← (reg)						
	(SP) ← (SP) - 2						
	((SP) + 1 : (SP)) ← (sreg)						
	(mem) ← ((SP) + 1 : (SP))						
	(SP) ← (SP) + 2						
	(reg) ← ((SP) + 1 : (SP))						
	(SP) ← (SP) + 2						
	(sreg) ← ((SP) + 1 : (SP))						
	(SP) ← (SP) + 2						
	(reg) ↔ (reg)						
	(reg) ↔ (mem)						
	(reg16) ↔ (AX)						

● アキュムレータ転送命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
IN	acc,imm8	1 1 1 0 0 1 0 W		2	10	
	acc,DX	1 1 1 0 1 1 0 W		1	8	
OUT	imm8,acc	1 1 1 0 0 1 1 W		2	10	
	DX,acc	1 1 1 0 1 1 1 W		1	8	
XLAT	source -table	1 1 0 1 0 1 1 1		1	11	

● アドレス・オブジェクト転送命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
LEA	reg16, mem16	1 0 0 0 1 1 0 1	mod reg r/m	2-4	2+E A	
LDS	reg16, mem32	1 1 0 0 0 1 0 1	mod reg r/m	2-4	16+E A	
LES	reg16, mem32	1 1 0 0 0 1 0 0	mod reg r/m	2-4	16+E A	

● フラグ・レジスタ転送命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
LAHF		1 0 0 1 1 1 1 1		1	4	
SAHF		1 0 0 1 1 1 1 0		1	4	
PUSHF		1 0 0 1 1 1 0 0		1	10	
POPF		1 0 0 1 1 1 0 1		1	8	

	オペレーション	フラグ					
		A	C	O	P	S	Z
	ifW=0(AL)←(imm8)						
	ifW=1(AX)←(imm8+1 : imm8)						
	ifW=0(AL)←((DX))						
	ifW=1(AX)←((DX)+1 : (DX))						
	ifW=0(imm8)←(AL)						
	ifW=1(imm8+1 : imm8)←(AX)						
	ifW=0((DX))←(AL)						
	ifW=1((DX)+1 : (DX))←(AX)						
	(AL)←((BX)+(AL))						

	オペレーション	フラグ					
		A	C	O	P	S	Z
	(reg16)←mem16						
	(reg16)←(mem32)						
	(DS)←(mem32+2)						
	(reg16)←(mem32)						
	(ES)←(mem32+2)						

	オペレーション	フラグ					
		A	C	O	P	S	Z
	(AH)←(SF):(ZF):X:(AF):X:(PF):X:(CF)						
	(SF):(ZF):X:(AF):X:(PF):X:(CF)←(AH)	×	×		×	×	×
	(SP)←(SP)-2						
	((SP)+1:(SP))←FLAGS						
	FLAGS←((SP)+1:(SP))	R	R	R	R	R	R
	(SP)←(SP)+2						

7.2 算術演算命令

◎加算命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
ADD	reg,reg	0 0 0 0 0 0 0 W	1 1 reg r/m	2	3	
	mem,reg	0 0 0 0 0 0 0 W	mod reg r/m	2-4	16+EA	
	reg,mem	0 0 0 0 0 0 1 W	mod reg r/m	2-4	9+EA	
	reg,imm	1 0 0 0 0 0 S W	1 1 0 0 0 r/m	3-4	4	
	mem,imm	1 0 0 0 0 0 S W	mod 0 0 0 r/m	3-6	17+EA	
	acc,imm	0 0 0 0 0 1 0 W		2-3	4	
ADC	reg,reg	0 0 0 1 0 0 0 W	1 1 reg r/m	2	3	
	mem,reg	0 0 0 1 0 0 0 W	mod reg r/m	2-4	16+EA	
	reg,mem	0 0 0 1 0 0 1 W	mod reg r/m	2-4	9+EA	
	reg,imm	1 0 0 0 0 0 S W	1 1 0 1 0 r/m	3-4	4	
	mem,imm	1 0 0 0 0 0 S W	mod 0 1 0 r/m	3-6	17+EA	
	acc,imm	0 0 0 1 0 1 0 W		2-3	4	
INC	reg8	1 1 1 1 1 1 1 W	1 1 0 0 0 r/m	2	3	
	mem	1 1 1 1 1 1 1 W	mod 0 0 0 r/m	2-4	15+EA	
	reg16	0 1 0 0 0 reg		1	2	
AAA		0 0 1 1 0 1 1 1		1	4	
DAA		0 0 1 0 0 1 1 1		1	4	

	オペレーション	フラグ					
		A	C	O	P	S	Z
	$(reg) \leftarrow (reg) + (reg)$	×	×	×	×	×	×
	$(mem) \leftarrow (mem) + (reg)$	×	×	×	×	×	×
	$(reg) \leftarrow (reg) + (mem)$	×	×	×	×	×	×
	$(reg) \leftarrow (reg) + data$	×	×	×	×	×	×
	$(mem) \leftarrow (mem) + data$	×	×	×	×	×	×
	ifW=0 (AL) $\leftarrow$ (AL)+data ifW=1 (AX) $\leftarrow$ (AX)+data	×	×	×	×	×	×
	$(reg) \leftarrow (reg) + (reg) + (CF)$	×	×	×	×	×	×
	$(mem) \leftarrow (mem) + (reg) + (CF)$	×	×	×	×	×	×
	$(reg) \leftarrow (reg) + (mem) + (CF)$	×	×	×	×	×	×
	$(reg) \leftarrow (reg) + data + (CF)$	×	×	×	×	×	×
	$(mem) \leftarrow (mem) + data + (CF)$	×	×	×	×	×	×
	ifW=0 AL $\leftarrow$ AL+data+(CF) ifW=1 AX $\leftarrow$ AX+data+(CF)	×	×	×	×	×	×
	$(reg8) \leftarrow (reg8) + 1$	×		×	×	×	×
	$(mem) \leftarrow (mem) + 1$	×		×	×	×	×
	$(reg16) \leftarrow (reg16) + 1$	×		×	×	×	×
	if((AL)&0FH)>9or(AF)=1then(AL) $\leftarrow$ (AL)+6, (AH) $\leftarrow$ (AH)+1, (AF) $\leftarrow$ 1, (CF) $\leftarrow$ (AF), (AL) $\leftarrow$ (AL)&0FH	×	×				
	if((AL)&0FH)>9or(AF)=1 then (AL) $\leftarrow$ (AL)+6, (CF) $\leftarrow$ (AF) $\vee$ (CF), (AF) $\leftarrow$ 1 if(AL)>9FHor(CF)=1 then (AL) $\leftarrow$ (AL)+60H, (CF) $\leftarrow$ 1	×	×		×	×	×

●減算命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
SUB	reg,reg	0 0 1 0 1 0 0 W	1 1 reg r/m	2	3	
	mem,reg	0 0 1 0 1 0 0 W	mod reg r/m	2-4	16+EA	
	reg,mem	0 0 1 0 1 0 1 W	mod reg r/m	2-4	9+EA	
	reg,imm	1 0 0 0 0 0 S W	1 1 1 0 1 r/m	3-4	4	
	mem,imm	1 0 0 0 0 0 S W	mod 1 0 1 r/m	3-6	17+EA	
	acc,imm	0 0 1 0 1 1 0 W		2-3	4	
SBB	reg,reg	0 0 0 1 1 0 0 W	1 1 reg r/m	2	3	
	mem,reg	0 0 0 1 1 0 0 W	mod reg r/m	2-4	16+EA	
	reg,mem	0 0 0 1 1 0 1 W	mod reg r/m	2-4	9+EA	
	reg,imm	1 0 0 0 0 0 S W	1 1 0 1 1 r/m	3-4	4	
	mem,imm	1 0 0 0 0 0 S W	mod 0 1 1 r/m	3-6	17+EA	
	acc,imm	0 0 0 1 1 1 0 W		2-3	4	
DEC	reg8	1 1 1 1 1 1 1 0	1 1 0 0 1 r/m	2	3	
	mem	1 1 1 1 1 1 1 W	mod 0 0 1 r/m	2-4	15+EA	
	reg16	0 1 0 0 1 reg		1	2	
NEG	reg	1 1 1 1 0 1 1 W	1 1 0 1 1 r/m	2	3	
	mem	1 1 1 1 0 1 1 W	mod 0 1 1 r/m	2-4	16+EA	
CMP	reg,reg	0 0 1 1 1 0 0 W	1 1 reg r/m	2	3	
	mem,reg	0 0 1 1 1 0 0 W	mod reg r/m	2-4	9+EA	
	reg,mem	0 0 1 1 1 0 1 W	mod reg r/m	2-4	9+EA	
	reg,imm	1 0 0 0 0 0 S W	1 1 1 1 1 r/m	3-4	4	
	mem,imm	1 0 0 0 0 0 S W	mod 1 1 1 r/m	3-6	10+EA	
	acc,imm	0 0 1 1 1 1 0 W		2-3	4	
AAS		0 0 1 1 1 1 1 1		1	4	
DAS		0 0 1 0 1 1 1 1		1	4	

	オペレーション	フラグ					
		A	C	O	P	S	Z
	(reg) ← (reg) - (reg)	×	×	×	×	×	×
	(mem) ← (mem) - (reg)	×	×	×	×	×	×
	(reg) ← (reg) - (mem)	×	×	×	×	×	×
	(reg) ← (reg) - data	×	×	×	×	×	×
	(mem) ← (mem) - data	×	×	×	×	×	×
	if W=0 (AL) ← (AL) - data if W=1 (AX) ← (AX) - data	×	×	×	×	×	×
	(reg) ← (reg) - (reg) - (CF)	×	×	×	×	×	×
	(mem) ← (mem) - (reg) - (CF)	×	×	×	×	×	×
	(reg) ← (reg) - (mem) - (CF)	×	×	×	×	×	×
	(reg) ← (reg) - data - (CF)	×	×	×	×	×	×
	(mem) ← (mem) - data - (CF)	×	×	×	×	×	×
	if W=0 (AL) ← (AL) - data - (CF) if W=1 (AX) ← (AX) - data - (CF)	×	×	×	×	×	×
	(reg8) ← (reg8) - 1	×		×	×	×	×
	(mem) ← (mem) - 1	×		×	×	×	×
	(reg16) ← (reg16) - 1	×		×	×	×	×
	(reg) ← 0 - (reg)	×	×	×	×	×	×
	(mem) ← 0 - (mem)	×	×	×	×	×	×
	(reg) - (reg)	×	×	×	×	×	×
	(mem) - (reg)	×	×	×	×	×	×
	(reg) - (mem)	×	×	×	×	×	×
	(reg) - data	×	×	×	×	×	×
	(mem) - data	×	×	×	×	×	×
	if W=0 (AL) - data if W=1 (AX) - data	×	×	×	×	×	×
	if ((AL)&0FH)>9 or (AF)=1 then (AL) ← (AL) - 6, (AH) ← (AH) - 1, (AF) ← 1 (CF) ← (AF), (AL) ← (AL) & 0FH	×	×	U	U	U	U
	if ((AL)&0FH)>9 or (AF)=1 then (AL) ← (AL) - 6, (CF) ← (AF) ∨ (CF), (AF) ← 1 if (AL)>9FH or (CF)=1 then (AL) ← (AL) - 60H, (CF) ← 1	×	×	U	×	×	×

7.3 乗除算命令

◎乗算命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
MUL	reg8	1 1 1 1 0 1 1 0	1 1 1 0 0 r/m	2	70-77	
	mem8	1 1 1 1 0 1 1 0	mod1 0 0 r/m	2-4	(76-83) + E A	
	reg16	1 1 1 1 0 1 1 1	1 1 1 0 0 r/m	2	118-133	
	mem16	1 1 1 1 0 1 1 1	mod1 0 0 r/m	2-4	(124-139) + E A	
IMUL	reg8	1 1 1 1 0 1 1 0	1 1 1 0 1 r/m	2	80-98	
	mem8	1 1 1 1 0 1 1 0	mod1 0 1 r/m	2-4	(86-104) + E A	
	reg16	1 1 1 1 0 1 1 1	1 1 1 0 1 r/m	2	128-154	
	mem16	1 1 1 1 0 1 1 1	mod1 0 1 r/m	2-4	(134-160) + E A	
AAM		1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0	2	83	

◎除算命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
DIV	reg8	1 1 1 1 0 1 1 0	1 1 1 1 0 r/m	2	80-90	

	オペレーション	フラグ					
		A	C	O	P	S	Z
	(AX)←(AL)×(reg8) EXT=AH if(EXT)=0then(CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
	(AX)←(AL)×(mem8) EXT=AH if(EXT)=0then(CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
	(DX:AX)←(AX)×(reg16) EXT=DX if(EXT)=0then (CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
	(DX:AX)←(AX)×(mem16) EXT=DX if(EXT)=0then (CF)←0 else (CF)←1;(OF)←(CF)	U	×	×	U	U	U
	(AX)←(AL)×(reg8) EXT=AH, LOW=AL if(EXT)=sign extension of(LOW)then(CF)←0 else (CF)←1;(OF)←(CF)	U	×	×	U	U	U
	(AX)←(AL)×(mem8) EXT=AH, LOW=AL if(EXT)=sign extension of(LOW)then(CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
	(DX:AX)←(AX)×(reg16) EXT=DX, LOW=AX if(EXT)=sign extension of(LOW)then(CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
	(DX:AX)←(AX)×(mem16) EXT=DX, LOW=AX if(EXT)=sign extension of(LOW)then(CF)←0 else(CF)←1;(OF)←(CF)	U	×	×	U	U	U
	(AH)←(AL)/0AH (AL)←(AL)%0AH	U	U	U	×	×	×

	オペレーション	フラグ					
		A	C	O	P	S	Z
	(temp)←(AX) if(temp)/(reg8)>FFH then (SP)←(SP)-2,((SP)+1:(SP))←FLAGS, (IF)←0,(TF)←0, (SP)←(SP)-2,((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)-2,((SP)+1:(SP))←(IP), (IP)←(0) else (AL)←(temp)/(reg8), (AH)←(temp)%(reg8)	U	U	U	U	U	U

	mem 8	1 1 1 1 0 1 1 0	mod 1 1 0 r/m	2 - 4	(86-96) + E A	
	reg16	1 1 1 1 0 1 1 1	1 1 1 1 0 r/m	2	144-162	
	mem16	1 1 1 1 0 1 1 1	mod 1 1 0 r/m	2 - 4	(150-168) + E A	
I D I V	reg8	1 1 1 1 0 1 1 0	1 1 1 1 1 r/m	2	101-112	
	mem 8	1 1 1 1 0 1 1 0	mod 1 1 1 r/m	2 - 4	(107-118) + E A	
	reg16	1 1 1 1 0 1 1 1	1 1 1 1 1 r/m	2	165-184	
	mem16	1 1 1 1 0 1 1 1	mod 1 1 1 r/m	2 - 4	(171-190) + E A	
A A D		1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0	2	60	
C B W		1 0 0 1 1 0 0 0		1	2	
C W D		1 0 0 1 1 0 0 1		1	5	

(temp)←(AX) if(temp)/(mem8)>FFH then (SP)←(SP)-2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(0) else (AL)←(temp)/(mem8),(AH)←(temp)%(mem8)	U	U	U	U	U	U
(temp)←(DX:AX) if(temp)/(reg16)>FFFFH then (SP)←(SP)-2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(0) else (AX)←(temp)/(reg16),(DX)←(temp)%(reg16)	U	U	U	U	U	U
(temp)←(DX:AX) if(temp)/(mem16)>FFFFH then (SP)←(SP)-2,((SP)+1:(SP))←FLAGS,IF←0,TF←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(0) else (AX)←(temp)/(mem16),(DX)←(temp)%(mem16)	U	U	U	U	U	U
(temp)←(AX) if (temp)/(reg8)>0 and (temp)/(reg8) >7FH or (temp)/(reg8)>0 and (temp)/(reg8) <0-7FH-1 then (SP)←(SP)-2,(SP)+1:(SP)←FLAGS, (IF)←0,(TF)←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS), (CS)←(2) (SP)←(SP)-2,((SP)+1:(SP))←(IP), (IP)←(0) else (AL)←(temp)/(reg8), (AH)←(temp)%(reg8)	U	U	U	U	U	U
(temp)←(AX) if (temp)/(mem8)>0 and (temp)/(mem8) >7FH or (temp)/(mem8)>0 and (temp)/(mem8) <0-7FH-1 then (SP)←(SP)-2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(0) else (AL)←(temp)/(mem8),(AH)←(temp)%(mem8)	U	U	U	U	U	U
(temp)←(DX:AX) if (temp)/(reg16)>0 and (temp)/(reg16) >7FFFH or (temp)/(reg16)<0 and (temp)/(reg16) <0-7FFFH-1 then (SP)←(SP)-2,((SP)+1:(SP))←FLAGS,(IF)←0,(TF)←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(0) else (AX)←(temp)/(reg16),(DX)←(temp)%(reg16)	U	U	U	U	U	U
(temp)←(DX:AX) if (temp)/(mem16)>0 and (temp)/(mem16) >7FFFH or (temp)/(mem16)<0 and (temp)/(mem16) <0-7FFFH-1 then (SP)←(SP)-2,((SP)+1:(SP))←FLAGS,IF←0,TF←0 (SP)←(SP)-2,((SP)+1:(SP))←(CS),(CS)←(2) (SP)←(SP)-2,((SP)+1:(SP))←(IP),(IP)←(0) else (AX)←(temp)/(mem16),(DX)←(temp)%(mem16)	U	U	U	U	U	U
(AL)←(AH)*0AH+(AL) (AH)←0	U	U	U	×	×	×
if(AL)<80H then(AH)←0 else (AH)←FFH						
if(AX)<8000H then(DX)←0 else (DX)←FFFFH						

7.4 論理演算命令

◎シングル・オペランド論理命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
NOT	reg	1 1 1 1 0 1 1 W	1 1 0 1 0 r/m	2	3	
	mem	1 1 1 1 0 1 1 W	mod 0 1 0 r/m	2 - 4	16 + E A	
SHL SAL	reg, 1	1 1 0 1 0 0 0 W	1 1 1 0 0 r/m	2	2	
	mem, 1	1 1 0 1 0 0 0 W	mod 1 0 0 r/m	2 - 4	15 + E A	
	reg, CL	1 1 0 1 0 0 1 W	1 1 1 0 0 r/m	2	8 + 4/bit	
	mem, CL	1 1 0 1 0 0 1 W	mod 1 0 0 r/m	2 - 4	20 + E A + 4/bit	
SHR	reg, 1	1 1 0 1 0 0 0 W	1 1 1 0 1 r/m	2	2	
	mem, 1	1 1 0 1 0 0 0 W	mod 1 0 1 r/m	2 - 4	15 + E A	
	reg, CL	1 1 0 1 0 0 1 W	1 1 1 0 1 r/m	2	8 + 4/bit	
	mem, CL	1 1 0 1 0 0 1 W	mod 1 0 1 r/m	2 - 4	20 + E A + 4/bit	
SAR	reg, 1	1 1 0 1 0 0 0 W	1 1 1 1 1 r/m	2	2	
	mem, 1	1 1 0 1 0 0 0 W	mod 1 1 1 r/m	2 - 4	15 + E A	
	reg, CL	1 1 0 1 0 0 1 W	1 1 1 1 1 r/m	2	8 + 4/bit	

オペレーション	フラグ					
	A	C	O	P	S	Z
ifW=0 (REG8)←FFH-(REG8) ifW=1 (REG16)←FFFFH-(REG16) ifW=0 (me8)←FFH-(mem8) ifW=1 (mem16)←FFFFH-(mem16)						
(CF)←high-order bit of(reg), (r/m)←(reg)×2 if high-order bit of (reg)≠(CF) then(OF)←1 else(OF)←0	U	×	×	×	×	×
(CF)←high-order bit of(mem), (mem)←(mem)×2 if high-order bit of (EA)≠(CF)then(OF)←1 else(OF)←0	U	×	×	×	×	×
(temp)←(CL), do while(temp)≠0 (CF)←high-order bit of(reg), (reg)←(reg)×2 (temp)←(temp)-1 (OF)undefined	U	×	U	×	×	×
(temp)←(CL), do while(temp)≠0 (CF)←high-order bit of(mem), (mem)←(mem)×2 (temp)←(temp)-1 (OF)undefined	U	×	U	×	×	×
(CF)←low-order bit of(reg), (reg)←(reg)/2 if high-order bit of (reg)≠next-to-high-order bit of (reg) then(OF)←1 else(OF)←0	U	×	×	×	×	×
(CF)←low-order bit of(mem), (mem)←(mem)/2 if high-order bit of (mem)≠next-to-high-order bit of(mem)then(OF)←1 slse(OF)←0	U	×	×	×	×	×
(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(reg), (reg)←(reg)/2 (temp)←(temp)-1 (OF)undefined	U	×	U	×	×	×
(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(mem), (mem)←(mem)/2 (temp)←(temp)-1 (OF)undefined	U	×	U	×	×	×
(CF)←low-order bit of(reg), (reg)←(reg)/2, (OF)←0	U	×	0	×	×	×
(CF)←low-order bit of(mem), (mem)←(mem)/2, (OF)←0	U	×	0	×	×	×
(temp)←(CL) do while(temp)≠0 (CF)←low-order bit of(reg), (reg)←(reg)/2 (temp)←(temp)-1 (OF)undefined	U	×	U	×	×	×

	mem,CL	1 1 0 1 0 0 1 W	mod 1 1 1 r/m	2 - 4	20 + E A + 4/bit	
R O L	reg,1	1 1 0 1 0 0 0 W	1 1 0 0 0 r/m	2	2	
	mem,1	1 1 0 1 0 0 0 W	mod 0 0 0 r/m	2 - 4	15 + E A	
	reg,CL	1 1 0 1 0 0 1 W	1 1 0 0 0 r/m	2	8 + 4/bit	
	mem,CL	1 1 0 1 0 0 1 W	mod 0 0 0 r/m	2 - 4	20 + E A + 4/bit	
R O R	reg,1	1 1 0 1 0 0 0 W	1 1 0 0 1 r/m	2	2	
	mem,1	1 1 0 1 0 0 0 W	mod 0 0 1 r/m	2 - 4	15 + E A	
	reg,CL	1 1 0 1 0 0 1 W	1 1 0 0 1 r/m	2	8 + 4/bit	
	mem,CL	1 1 0 1 0 0 1 W	mod 0 0 1 r/m	2 - 4	20 + E A + 4/bit	
R C L	reg,1	1 1 0 1 0 0 0 W	1 1 0 1 0 r/m	2	2	
	mem,1	1 1 0 1 0 0 0 W	mod 0 1 0 r/m	2 - 4	15 + E A	
	reg,CL	1 1 0 1 0 0 1 W	1 1 0 1 0 r/m	2	8 + 4/bit	

$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(CF) \leftarrow$ low-order bit of (mem) , $(mem) \leftarrow (mem)/2$ $(temp) \leftarrow (temp) - 1$ (OF) undefined	U	×	U	×	×	×
$(CF) \leftarrow$ high-order bit of (reg) , $(reg) \leftarrow (reg) * 2 + (CF)$ if high-order bit of $(reg) \neq (CF)$ then $(OF) \leftarrow 1$ else $(OF) \leftarrow 0$		×	×			
$(CF) \leftarrow$ high-order bit of (mem) , $(mem) \leftarrow (mem) * 2 + (CF)$ if high-order bit of $(mem) \neq (CF)$ then $(OF) \leftarrow 1$ else $(OF) \leftarrow 0$		×	×			
$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(CF) \leftarrow$ high-order bit of (reg) , $(reg) * 2 + (CF)$ $(temp) \leftarrow (temp) - 1$ (OF) undefined		×	U			
$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(CF) \leftarrow$ high-order bit of (mem) , $(mem) \leftarrow (mem) * 2 + (CF)$ $(temp) \leftarrow (temp) - 1$ (OF) undefined		×	U			
$(CF) \leftarrow$ low-order bit of (reg) , $(reg) \leftarrow (reg)/2$ high-order bit of $(reg) \leftarrow (CF)$ if high-order bit of $(reg) \neq$ next-to-high-order bit of (reg) then $(OF) \leftarrow 1$ else $(OF) \leftarrow 0$		×	×			
$(CF) \leftarrow$ low-order bit of (mem) , $(mem) \leftarrow (mem)/2$ high-order bit of $(mem) \leftarrow (CF)$ if high-order bit of $(mem) \neq$ next-to-high-order bit of (mem) then $(OF) \leftarrow 1$ else $(OF) \leftarrow 0$		×	×			
$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(CF) \leftarrow$ low-order bit of (reg) , $(reg) \leftarrow (reg)/2$ high-order bit of $(reg) \leftarrow (CF)$ $(temp) \leftarrow (temp) - 1$ (OF) undefined		×	U			
$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(CF) \leftarrow$ low-order bit of (mem) , $(mem) \leftarrow (mem)/2$ high-order bit of $(mem) \leftarrow (CF)$ $(temp) \leftarrow (temp) - 1$ (OF) undefined		×	U			
$(tmpcf) \leftarrow (CF)$, $(CF) \leftarrow$ high-order bit of (reg) $(reg) \leftarrow (reg) * 2 + (tmpcf)$ if high-order bit of $(reg) \leftarrow (CF)$ then $(OF) \leftarrow 1$ else $(OF) \leftarrow 0$		×	×			
$(tmpcf) \leftarrow (CF)$, $(CF) \leftarrow$ high-order bit of (mem) $(mem) \leftarrow (mem) * 2 + (tmpcf)$ if high-order bit of $(mem) \neq (CF)$ then $(OF) \leftarrow 1$ else $(OF) \leftarrow 0$		×	×			
$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(tmpcf) \leftarrow (CF)$, $(CF) \leftarrow$ high-order bit of (reg) $(reg) \leftarrow (reg) * 2 + (tmpcf)$ $(temp) \leftarrow (temp) - 1$ (OF) unde fined		×	U			

	mem,CL	1 1 0 1 0 0 1 W	mod 0 1 0 r/m	2 - 4	20 + E A + 4/bit
R C R	reg,l	1 1 0 1 0 0 0 W	1 1 0 1 1 r/m	2	2
	mem,l	1 1 0 1 0 0 0 W	mod 0 1 1 r/m	2 - 4	15 + E A
	reg,CL	1 1 0 1 0 0 1 W	1 1 0 1 1 r/m	2	8 + + 4/bit
	mem,CL	1 1 0 1 0 0 1 W	mod 0 1 1 r/m	2 - 4	20 + E A + 4/bit

$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(tmpcf) \leftarrow (CF), (CF) \leftarrow \text{high-order bit of } (mem)$ $(mem) \leftarrow (mem) * 2 + (tmpcf)$ $(temp) \leftarrow (temp) - 1$ (OF)undefined		×	U				
$(tmpcf) \leftarrow (CF),$ $(CF) \leftarrow \text{low-order bit of } (reg), (reg) \leftarrow (reg) / 2$ $\text{high-order bit of } (reg) \leftarrow (tmpcf)$ if high-order bit of $(reg) \neq \text{next-to-high-order bit of } (reg)$ then $(OF) \leftarrow 1$ else $(OF) \leftarrow 0$		×	×				
$(tmpcf) \leftarrow (CF),$ $(CF) \leftarrow \text{low-order bit of } (mem), (mem) \leftarrow (mem) / 2$ $\text{high-order bit of } (mem) \leftarrow (tmpcf)$ if high-order bit of $(mem) \neq \text{next-to-high-order bit of } (mem)$ then $(OF) \leftarrow 1$ else $(OF) \leftarrow 0$		×	×				
$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(tmpcf) \leftarrow (CF), (CF) \leftarrow \text{low-order bit of } (reg)$ $(reg) \leftarrow (reg) / 2$ $\text{high-order bit of } (r/m) \leftarrow (tmpcf)$ $(temp) \leftarrow (temp) - 1$ (OF)undefined		×	U				
$(temp) \leftarrow (CL)$ do while $(temp) \neq 0$ $(tmpcf) \leftarrow (CF), (CF) \leftarrow \text{low-order bit of } (mem)$ $(mem) \leftarrow (mem) / 2$ $\text{high-order bit of } (mem) \leftarrow (tmpcf)$ $(temp) \leftarrow (temp) - 1$ (OF)undefined		×	U				

◎ダブル・オペランド論理命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
AND	reg,reg	0 0 1 0 0 0 0 W	1 1 reg r/m	2	3	
	mem,reg	0 0 1 0 0 0 0 W	mod reg r/m	2-4	16+E A	
	reg,mem	0 0 1 0 0 0 1 W	mod reg r/m	2-4	9+E A	
	reg,imm	1 0 0 0 0 0 0 W	1 1 1 0 0 r/m	3-4	4	
	mem,imm	1 0 0 0 0 0 0 W	mod 1 0 0 r/m	3-6	17+E A	
	acc,imm	0 0 1 0 0 1 0 W		2-3	4	
TEST	reg,reg	1 0 0 0 0 1 0 W	1 1 reg r/m	2	3	
	mem,reg	1 0 0 0 0 1 0 W	mod reg r/m	2-4	9+E A	
	reg,imm	1 1 1 1 0 1 1 W	1 1 0 0 0 r/m	3-4	5	
	mem,imm	1 1 1 1 0 1 1 W	mod 0 0 0 r/m	3-6	11+E A	
	acc,imm	1 0 1 0 1 0 0 W		2-3	4	
OR	reg,reg	0 0 0 0 1 0 0 W	1 1 reg r/m	2	3	
	mem,reg	0 0 0 0 1 0 0 W	mod reg r/m	2-4	16+E A	
	reg,mem	0 0 0 0 1 0 1 W	mod reg r/m	2-4	9+E A	
	reg,imm	1 0 0 0 0 0 0 W	1 1 0 0 1 r/m	3-4	4	
	mem,imm	1 0 0 0 0 0 0 W	mod 0 0 1 r/m	3-6	17+E A	
	acc,imm	0 0 0 0 1 1 0 W		2-3	4	
XOR	reg,reg	0 0 1 1 0 0 0 W	1 1 reg r/m	2	3	
	mem,reg	0 0 1 1 0 0 0 W	mod reg r/m	2-4	16+E A	
	reg,mem	0 0 1 1 0 0 1 W	mod reg r/m	2-4	9+E A	
	reg,imm	1 0 0 0 0 0 0 W	1 1 1 1 0 r/m	3-4	4	
	mem,imm	1 0 0 0 0 0 0 W	mod 1 1 0 r/m	3-6	17+E A	
	acc,imm	0 0 1 1 0 1 0 W		2-3	4	

	オペレーション	フラグ					
		A	C	O	P	S	Z
	$(reg) \leftarrow (reg) \& (reg)$	U	0	0	×	×	×
	$(mem) \leftarrow (mem) \& (reg)$	U	0	0	×	×	×
	$(reg) \leftarrow (reg) \& (mem)$	U	0	0	×	×	×
	$(reg) \leftarrow (reg) \& data$	U	0	0	×	×	×
	$(mem) \leftarrow (mem) \& data$	U	0	0	×	×	×
	ifW=0 (AL) $\leftarrow$ (AL)& data ifW=1 (AX) $\leftarrow$ (AX)& data	U	0	0	×	×	×
	$(reg) \& (reg)$	U	0	0	×	×	×
	$(reg) \& (mem)$	U	0	0	×	×	×
	$(reg) \& data$	U	0	0	×	×	×
	$(mem) \& data$	U	0	0	×	×	×
	ifW=0 (AL)& data ifW=1 (AX)& data	U	0	0	×	×	×
	$(reg) \leftarrow (reg) \vee (reg)$	U	0	0	×	×	×
	$(mem) \leftarrow (mem) \vee (reg)$	U	0	0	×	×	×
	$(reg) \leftarrow (reg) \vee (mem)$	U	0	0	×	×	×
	$(reg) \leftarrow (reg) \vee data$	U	0	0	×	×	×
	$(mem) \leftarrow (mem) \vee data$	U	0	0	×	×	×
	ifW=0 (AL) $\leftarrow$ (AL) $\vee$ data ifW=1 (AX) $\leftarrow$ (AX) $\vee$ data	U	0	0	×	×	×
	$(reg) \leftarrow (reg) \vee (reg)$	U	0	0	×	×	×
	$(mem) \leftarrow (mem) \vee (reg)$	U	0	0	×	×	×
	$(reg) \leftarrow (reg) \vee (mem)$	U	0	0	×	×	×
	$(reg) \leftarrow (reg) \vee data$	U	0	0	×	×	×
	$(mem) \leftarrow (mem) \vee data$	U	0	0	×	×	×
	ifW=0 (AL) $\leftarrow$ (AL) $\vee$ data ifW=1 (AX) $\leftarrow$ (AX) $\vee$ data	U	0	0	×	×	×

7.5 スtring操作命令

◎リピート・プリフィクス命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
REP REPE REPZ		1 1 1 1 0 0 1 1		1	2	
REPNE REPNZ		1 1 1 1 0 0 1 0		1	2	

◎プリミティブ・String命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
MOVS MOVSB* MOVSW*	dst- string, src- string	1 0 1 0 0 1 0 W		1	$\frac{9}{17} + \frac{\text{rep}}{18}$	
CMPS	dst- string, src- string	1 0 1 0 0 1 1 W		1	$\frac{9}{22} + \frac{\text{rep}}{22}$	
SCAS	dst- string	1 0 1 0 1 1 1 W		1	$\frac{9}{15} + \frac{\text{rep}}{15}$	
LODS	src- string	1 0 1 0 1 1 0 W		1	$\frac{9}{13} + \frac{\text{rep}}{12}$	
STOS	dst- string	1 0 1 0 1 0 1 W		1	$\frac{9}{10} + \frac{\text{rep}}{11}$	

オペレーション	フ ラ グ					
	A	C	O	P	S	Z
do while (CX)≠0 続くバイトのプリミティブ・ストリング命令を実行する (CX)←(CX)-1 保留割込みがあれば処理する プリミティブ命令がCMPSまたはSCASでかつ、(ZF)≠1のときループを抜出る。						
do while (CX)≠0 続くバイトのプリミティブ・ストリング命令を実行する (CX)←(CX)-1 保留割込みがあれば処理する (ZF)≠0 のときループを抜出る						

オペレーション	フ ラ グ					
	A	C	O	P	S	Z
ifW = 0 ((DI))←((SI)) if(DF)=0 then (SI)←(SI)+1, (DI)←(DI)+1 else(SI)←(SI)-1, (DI)←(DI)-1 ifW=1 ((DI)+1 : (DI))←((SI)+1 : (SI)) if(DF)=0 then(SI)←(SI)+2, (DI)←(DI)+2 else(SI)←(SI)-2, (DI)←(DI)-2						
ifW=0 ((SI))←((DI)) if(DF)=0 then(SI)←(SI)+1, (DI)←(DI)+1 else(SI)←(SI)-1, (DI)←(DI)-1 ifW=1 ((SI)+1 : (SI))←((DI)+1 : (DI)) if(DF)=0 then(SI)←(SI)+2, (DI)←(DI)+2 else(SI)←(SI)-2, (DI)←(DI)-2	×	×	×	×	×	×
ifW=0 (AL)←((DI)) if(DF)=0 then(DI)←(DI)+1 else(DI)←(DI)-1 ifW=1 (AX)←((DI)+1 : (DI)) if(DF)=0 then(DI)←(DI)+2 else(DI)←(DI)-2	×	×	×	×	×	×
ifW=0 (AL)←((SI)) if(DF)=0 then(SI)←(SI)+1 else(SI)←(SI)-1 ifW=1 (AX)←((SI)+1 : (SI)) if(DF)=0 then(SI)←(SI)+2 else(SI)←(SI)-2						
ifW=0 ((DI))←(AL) if(DF)=0 then(DI)←(DI)+1 else(DI)←(DI)-1 ifW=1 ((DI)+1 : (DI))←(AX) if(DF)=0 then(DI)←(DI)+2 else(DI)←(DI)-2						

7.6 分岐命令

◎制御転送命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
CALL	near- proc	1 1 1 0 1 0 0 0		3	19	
	regptr16	1 1 1 1 1 1 1 1	1 1 0 1 0 r/m	2	16	
	memptr 16	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	2-4	21+E A	
	far- proc	1 0 0 1 1 0 1 0		5	28	
	memptr 32	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	2-4	37+E A	
JMP	near- label	1 1 1 0 1 0 0 1		3	15	
	short- label	1 1 1 0 1 0 1 1		2	15	
	regptr16	1 1 1 1 1 1 1 1	1 1 1 0 0 r/m	2	11	
	memptr 16	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	2-4	18+E A	
	far- label	1 1 1 0 1 0 1 0		5	15	
	memptr 32	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	2-4	24+E A	
RET		1 1 0 0 0 0 1 1		1	8	
	pop- value	1 1 0 0 0 0 1 0		3	12	
		1 1 0 0 1 0 1 1		1	18	
	pop- value	1 1 0 0 1 0 1 0		3	17	

	オペレーション	フラグ					
		A	C	O	P	S	Z
	(SP)←(SP)-2 ((SP)+1:(SP))←(IP), (IP)←(IP)+disp						
	(SP)←(SP)-2 ((SP)+1:(SP))←(IP), (IP)←(regptr16)						
	(SP)←(SP)-2 ((SP)+1:(SP))←(IP), (IP)←(memptr16)						
	(SP)←(SP)-2 ((SP)+1:(SP))←(CS), (CS)←seg (SP)←(SP)-2 ((SP)+1:(SP))←(IP), (IP)←offset						
	(SP)←(SP)-2 ((SP)+1:(SP))←(CS), (CS)←(mmptr32+2) (SP)←(SP)-2 ((SP)+1:(SP))←(IP), (IP)←(memptr32)						
	(IP)←(IP)+disp						
	(IP)←(IP)+disp						
	(IP)←(regptr16)						
	(IP)←(memptr16)						
	(CS)←seg (IP)←offset						
	(CS)←(memptr+2) (IP)←(memptr32)						
	(IP)←((SP)+1:(SP)) (SP)←(SP)+2						
	(IP)←((SP)+1:(SP)) (SP)←(SP)+2, (SP)←(SP)+data						
	(IP)←((SP)+1:(SP)) (SP)←(SP)+2 (CS)←((SP)+1:(SP)) (SP)←(SP)+2						
	(IP)←((SP)+1:(SP)) (SP)←(SP)+2 (CS)←((SP)+1:(SP)) (SP)←(SP)+2, (SP)←(SP)+data						

◎条件付制御転送命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
J E J Z	short- label	0 1 1 1 0 1 0 0		2	16 / 4	
J L J N G E	"	0 1 1 1 1 1 0 0		2	16 / 4	
J L E J N G	"	0 1 1 1 1 1 1 0		2	16 / 4	
J B J N A E	"	0 1 1 1 0 0 1 0		2	16 / 4	
J B E J N A	"	0 1 1 1 0 1 1 0		2	16 / 4	
J P J P E	"	0 1 1 1 1 0 1 0		2	16 / 4	
J O	"	0 1 1 1 0 0 0 0		2	16 / 4	
J S	"	0 1 1 1 1 0 0 0		2	16 / 4	
J N E J N Z	"	0 1 1 1 0 1 0 1		2	16 / 4	
J N L J G E	"	0 1 1 1 1 1 0 1		2	16 / 4	
J N L E J G	"	0 1 1 1 1 1 1 1		2	16 / 4	
J N B J A E	"	0 1 1 1 0 0 1 1		2	16 / 4	
J N B E J A	"	0 1 1 1 0 1 1 1		2	16 / 4	
J N P J P O	"	0 1 1 1 1 0 1 1		2	16 / 4	
J N O	"	0 1 1 1 0 0 0 1		2	16 / 4	
J N S	"	0 1 1 1 1 0 0 1		2	16 / 4	

◎くり返し転送命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
LOOP	"	1 1 1 0 0 0 1 0		2	17 / 5	
LOOPZ LOOPE	"	1 1 1 0 0 0 0 1		2	18 / 6	
LOOPNZ LOOPNE	"	1 1 1 0 0 0 0 0		2	19 / 5	
J C X Z	"	1 1 1 0 0 0 1 1		2	18 / 6	

	オ ペ レ ー シ ョ ン	フ ラ グ					
		A	C	O	P	S	Z
	if(ZF)=1 (IP)←(IP)+disp						
	if(SF)∨(OF)=1 (IP)←(IP)+disp						
	if [(SF)∨(OF)] ∨(ZF)=1 (IP)←(IP)+disp						
	if(CF)=1 (IP)←(IP)+disp						
	if(CF)∨(ZF)=1 (IP)←(IP)+disp						
	if(PF)=1 (IP)←(IP)+disp						
	if(OF)=1 (IP)←(IP)+disp						
	if(SF)=1 (IP)←(IP)+disp						
	if(ZF)=0 (IP)←(IP)+disp						
	if(SF)∨(OF)=0 (IP)←(IP)+disp						
	if [(SF)∨(OF)] ∨(ZF)=0 (IP)←(IP)+disp						
	if(CF)=0 (IP)←(IP)+disp						
	if(CF)∨(ZF)=0 (IP)←(IP)+disp						
	if(PF)=0 (IP)←(IP)+disp						
	if(OF)=0 (IP)←(IP)+disp						
	if(SF)=0 (IP)←(IP)+disp						

	オ ペ レ ー シ ョ ン	フ ラ グ					
		A	C	O	P	S	Z
	(CX)←(CX)-1						
	if(CX)≠0 (IP)←(IP)+disp						
	(CX)←(CX)-1						
	if(ZF)=1 and(CX)≠0 (IP)←(IP)+disp						
	(CX)←(CX)-1						
	if(ZF)=0 and(CX)≠0 (IP)←(IP)+disp						
	if(CX)=0 (IP)←(IP)+disp						

◎割り込み転送命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
INT	3	1 1 0 0 1 1 0 0		1	52	
	imm8 (≠3)	1 1 0 0 1 1 0 1		2	51	
INTO		1 1 0 0 1 1 1 0		1	53 / 4	
IRET		1 1 0 0 1 1 1 1		1	24	

	オペレーション	フ ラ グ					
		A	C	O	P	S	Z
	$(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow \text{FLAGS}, (IF) \leftarrow 0, (TF) \leftarrow 0$ $(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow (CS), (CS) \leftarrow (14)$ $(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow (IP), (IP) \leftarrow (12)$						
	$(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow \text{FLAGS}, (IF) \leftarrow 0, (TF) \leftarrow 0$ $(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow (CS), (CS) \leftarrow (\text{type} \times 4 + 2)$ $(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow (IP), (IP) \leftarrow (\text{type} \times 4)$						
	$\text{if}(\text{OF}) = 1$ $(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow \text{FLAGS}, (IF) \leftarrow 0, (TF) \leftarrow 0$ $(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow (CS), (CS) \leftarrow (12H)$ $(SP) \leftarrow (SP) - 2$ $((SP) + 1 : (SP)) \leftarrow (IP), (IP) \leftarrow (10H)$						
	$(IP) \leftarrow ((SP) + 1 : (SP)), (SP) \leftarrow (SP) + 2$ $(CS) \leftarrow ((SP) + 1 : (SP)), (SP) \leftarrow (SP) + 2$ $\text{FLAGS} \leftarrow ((SP) + 1 : (SP)), (SP) \leftarrow (SP) + 2$	×	×	×	×	×	×

7.7 CPU制御命令

◎フラグ制御命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
CLC		1 1 1 1 1 0 0 0		1	2	
CMC		1 1 1 1 0 1 0 1		1	2	
STC		1 1 1 1 1 0 0 1		1	2	
CLD		1 1 1 1 1 1 0 0		1	2	
STD		1 1 1 1 1 1 0 1		1	2	
CLI		1 1 1 1 1 0 1 0		1	2	
STI		1 1 1 1 1 0 1 1		1	2	

◎CPU制御命令

ニーモニック	オペランド	オペレーション・コード		バイト 数	クロック 数	
		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0			
HLT		1 1 1 1 0 1 0 0		1	2	
WAIT		1 0 0 1 1 0 1 1		1	3 + 5n	
ESC	ext-op, reg	1 1 0 1 1 XXX	1 1 YYY r/m	2	2	
	ext-op, mem	1 1 0 1 1 XXX	modYYY r/m	2-4	8 + EA	
LOCK		1 1 1 1 0 0 0 0		1	2	
NOP		1 0 0 1 0 0 0 0		1	3	

	オペレーション	フラグ					
		A	C	O	P	S	Z
	(CF)←0		0				
	(CF)← $\overline{(\text{CF})}$		×				
	(CF)←1		1				
	(DF)←0						
	(DF)←1						
	(IF)←0						
	(IF)←1						

	オペレーション	フラグ					
		A	C	O	P	S	Z
	CPU halt						
	CPU wait						
	CPU escape data bus←(reg)						
	CPU escape data bus←(mem)						
	Bus Lock Prefix						
	ノー・オペレーション						

PC-9801アセンブリ言語プログラミング入門 I

著 者 川村 清 ©Kiyoshi Kawamura 1983

発行者 田村正隆

発行所 株式会社 ナツメ社
東京都千代田区神田神保町 1-52(〒101)
電話 03(291)1257(代表)
振替 東京 3-58661

印 刷 ラン印刷社

製 本 三修紙工

ISBN4-8163-0305-7

PC-8001 プログラミング入門

菅野 盈著 1200円

●パソコン入門のベストセラーです。プログラミングとは何かということが、この本によってわかったという読者の声がたくさん寄せられています。

【内容】 ◆ダイレクトモードって何? ◆画面に線を引いてみましょう ◆エラーなんかこわくない ◆フローチャートを知らなくても ◆画面の「黒板消し」 ◆カラーの棒グラフ ◆ボールを壁に向かって移動 ◆電話帳をカセットテープへ保存 ◆休日出勤手当の計算 ◆機械語プログラムの予備知識 ◆実務上に活用するために

PC-8001 ビジネス・プログラミング

谷口公三・菅野 盈共著 1200円

●本書はパソコンをビジネスにどう活用するか、その手順を販売管理を例にとりて解説したものです。むずかしいといわれているディスクのあつかいも、書いてあるとおりにキーインしていけば自然に理解できるようになっています。

【内容】 ◆プリンタの使い方 ◆ディスク用システムの起動 ◆ファイルについて ◆データファイル作成用プログラム ◆得意先マスタファイル ◆データの読み出しと実行 ◆ファイルの突き合わせ ◆商品コード・数量・単価の修正 ◆配列テーブルの検索 ◆チェックリストの印刷 ◆単純なランダムファイル ◆電話番号を加えたプログラム ◆INDEX作成のプログラム ◆実用的な乱処理プログラム ◆得意先別・製品別売上集計 ◆小計と合計の印刷プログラム

PC-8001 徹底解析

新藤義昭著 1600円

●コンピュータシステムの基礎構造を理解し、PC-8001をより高いレベルで応用していくためのハンドブックです。

【内容】 ◆オペレーティングシステムの目的 ◆OSとしてのN-BASIC ◆N-BASICの中間言語形式 ◆DISK-BASICのファイル構造 ◆PC-8001のI/Oマップ ◆キーボードの制御 ◆PC-8031のコマンドフォーマット ◆RS-232Cインターフェース ◆音響カブラ ◆通信端末と通信サブシステム ◆パーソナルコンピュータによる機械語開発 ◆アセンブラ ◆デバッグ ◆オブジェクトプログラムの実行 ◆コンピュータグラフィックス ◆音声認識と音声合成 ◆ローカルネットワーク

PC-8801 プログラミング入門

野々山隆幸著 1400円

●ズブの素人にも理解できる恰好の入門書です。漢字の使い方も解説してあります。

【内容】 ◆デモプログラムのロード ◆関数計算もできます ◆N88-BASICのスタート方法 ◆システムディスクのバックアップ ◆プログラムの作成 ◆プログラムのセーブ ◆経営分析プログラム ◆データインプットのプログラム ◆設備投資計画プログラム ◆棒グラフの作成 ◆エンド部分とサブルーチンのプログラム ◆利益計画プログラム ◆損益分岐点図表プログラム ◆予算実績差異分析プログラム ◆PC-8001用プログラムを使うには

PC-8801 ビジネス・プログラミング

布田 治著 1500円

●ビジネスへの適用の手順を販売管理を例にとってやさしく解説してあります。ディスクベースですので、ディスクをお持ちの読者向けです。ビジネス・ソフトの基本は、この本にすべて含まれています。

【内容】 ◆問題解決の定石 ◆ディスクシステム「ひらけゴマ」 ◆ディスクに何か書き込もう ◆入力プログラムの作成 ◆漢字で見出しを作ろう ◆項目の印刷方法アラカルト ◆どう並べかえるか ◆分類プログラムの実行 ◆マッチングプログラムの実行 ◆印刷文字のいろいろ ◆ランダムファイルってなに ◆書いたものは読めなきゃウソ ◆更新の「三種の神器」 ◆漢字名称はクセモノ ◆インデックスがなきゃランダムじゃないよ ◆分析表の作成 ◆いまやグラフィック時代 ◆顔は円グラフ ◆最後に笑うのはCHAINくん

PC-8801 アセンブリ言語プログラミング入門

脇 英世著 1600円

●内蔵アセンブラを使ったマシン語入門です。具体的な短いプログラムをサンプルに解説していますので、大変わかりやすくなっています。数値のソート、文字のソートまで扱っています。

【内容】 ◆BASICとアセンブリ言語 ◆予備知識はいりません ◆モニタプログラムの動かし方 ◆1と0をひっくり返すこと ◆大きい数をさがすこと ◆16ビットの足し算 ◆引き算はこうします ◆和を計算する ◆ASCII符号とはなんだろう ◆同じ文字列かどうかしらべる ◆精度の高い足し算 ◆簡単な並べかえ ◆キーワードとジャンプテーブルを使う ◆PC-8801のBASICを探検する ◆BASICと機械語のプログラムをつなげる ◆16ビットのデータを並べかえる ◆機械語プログラムの呼び出し方

PC-9801 プログラミング入門

野々山隆幸著 1500円

●ズブの素人にも理解できる恰好の入門書です。漢字の使い方も解説してあります。

【内容】 ◆デモプログラムのロード ◆関数計算もできます ◆N88-BASICのスタート方法 ◆システムディスクのバックアップ ◆プログラムの作成 ◆プログラムのセーブ ◆経営分析プログラム ◆データインプットのプログラム ◆設備投資計画プログラム ◆棒グラフの作成 ◆エンド部分とサブルーチンのプログラム ◆利益計画プログラム ◆損益分岐点図表プログラム ◆予算実績差異分析プログラム ◆PC-8001用プログラムを使うには

PC-9801 ビジネス・プログラミング

布田 治著 1500円

●ビジネスへの適用の手順を販売管理を例にとってやさしく解説してあります。ディスクベースですので、ディスクをお持ちの読者向けです。ビジネス・ソフトの基本は、この本にすべて含まれています。

【内容】 ◆問題解決の定石 ◆ディスクシステム「ひらけゴマ」 ◆ディスクに何か書き込もう ◆入力プログラムの作成 ◆漢字で見出しを作ろう ◆項目の印刷方法アラカルト ◆どう並べかえるか ◆分類プログラムの実行 ◆マッチングプログラムの実行 ◆印刷文字のいろいろ ◆ランダムファイルってなに ◆書いたものは読めなきゃウソ ◆更新の「三種の神器」 ◆漢字名称はクセモノ ◆インデックスがなきゃランダムじゃないよ ◆分析表の作成 ◆いまやグラフィック時代 ◆顔は円グラフ ◆最後に笑うのはCHAINくん

PC-9801 アセンブリ言語プログラミング入門

川村 清著 1600円

●16ビットマイコン8086のマシン語入門です。短いサンプルプログラムを数多くあげ、実行結果・入力方法などをディスプレイ表示にしてありますので、大変理解しやすくなっています。

【内容】 ◆PC-9801のOS ◆メモリのセグメンテーション ◆機械語モニタの起動方法 ◆メモリの内容をコピーするプログラム ◆メッセージを表示するプログラム ◆16ビットの加減算を行なうプログラム ◆16ビットデータを10倍するプログラム ◆サブルーチンの使い方 ◆サブルーチン・コールとレジスタ回避 ◆分岐のあるプログラム ◆くり返し処理のあるプログラム ◆簡単なデモンストレーション ◆8086のインストラクション活用表

PC-9801 グラフィックス入門

脇 英世著 1600円

●インテル8086を搭載した16ビット・パソコンでのグラフィックス入門です。グラフィックスに必要なコマンドをやさしく解説し、色の塗り方、簡単な図形の描き方を短いプログラムを折りこみながら説明しています。

【内容】 ◆座標系とは何んだらう ◆WINDOWとVIEWの使い方 ◆円はどうやって描くのだらう ◆可変後退翼ジェット機を作る ◆ラインスタイルとは何んだらう ◆ゴールデンゲートブリッジを作る ◆タイルパターンを作るためのツール ◆ペイント文で色を塗る ◆物体を回転させてみよう ◆3次元空間における拡大と縮小 ◆ワイヤー・フレーム・モデルを作る ◆宇宙船エンタープライズ号を作る

PC-6001 グラフィックス入門

矢野和代著 1200円

●ピコピコサウンドのゲームばかりではバビコンもかわいそう。これは、ちょっとノーブルに次元を変えて楽しむための本です。CRTディスプレイによるグラフィックスとプロッタプリンタを使ったグラフィックスをあつかっています。

【内容】 ◆バビコンでCGを ◆めくるめくページ ◆色、いろいろ ◆折れてもグラフィック ◆バビコン絵本 ◆ダイヤモンドの輝き ◆お知恵をはいしゃく ◆PC-6022はなんでも屋 ◆色を変える ◆あて名印刷プログラム ◆プロッタ、それは魔法の手 ◆文字のさかだち ◆くるくるくる、目がまわる ◆何がでてくればいいの? ◆交通整理ピーッ! ◆わかりやすく簡単に

PC-6001 プログラミング入門

安立隆一著 1200円

●バビコン操縦法から簡単なゲーム・プログラミングまでやさしく解説してあります。

【内容】 ◆スイッチ・オン ◆たし算をやってみよう ◆ついでに三角関数も ◆さあいよいよプログラム ◆LIST命令で、すっきり ◆プログラム、へんしーん ◆じゃま者は消せ! ◆ゲームのプログラムを作ろう ◆バビコンのサイコロ ◆何回であたったかな ◆ダイスゲームとは ◆サイコロの入れかえ ◆バビコンで音楽を ◆和音もできます ◆テンポも変えられる ◆SOUND命令を使ってみよう ◆UFOを射て ◆UFOメインプログラム ◆ビーム砲操作メインプログラム ◆バビコンとジャンケン ◆ジャンケンの手の形を作る ◆勝ち負けの判断 ◆バビコンの可能性

FM-7 プログラミング入門

● 6 月中旬発行

菅野 盈著 1200円

●パソコンの知識がまったくなくても、この本に書いてあるとおりにキーインしていけば、プログラミングとはどんなものなのかが自然に理解できるはずで、プログラムのテーマにも身近な例を使って解説しました。

【内容】 ◆デモテープを動かしてみよう ◆いよいよプログラムモード ◆プログラムのセーブ ◆プログラムの処理スピードを体得 ◆数あてゲーム ◆くりかえして遊びたいとき ◆もういくつ寝るとお正月 ◆まず人間が計算するとしたら ◆棒グラフプログラムのあらし ◆カラーの棒グラフ ◆ボールを壁にぶつけるプログラム ◆はねかえって元の位置へ ◆電話帳プログラム ◆給与テーブルの登録 ◆休日出勤手当一覧表 ◆楽しいシンセサイザ ◆モーツアルトのメヌエット ◆蒸気機関車のプラストの音 ◆漢字の画面表示

PC-8001MKⅡ プログラミング入門

● 6 月中旬発行

安立雅子著 1200円

●ベストセラーPC-8001と上位コンパチ機種の入門書です。基本的なプログラミングのほかに、強化されたグラフィック機能の解説も加えました。

【内容】 ◆スイッチ・オン ◆グラフィック文字と空白 ◆ちょっと本格派プログラム ◆エラーメッセージありがとう ◆スピードはどれくらい？ ◆ゲームのプログラムを作ろう ◆ダイスゲームとは ◆配列変数を使います ◆結果を表示 ◆UFOを射て ◆パソコンとジャンケン ◆トライアングル・ローテーション ◆ヒストグラムもこんなに簡単 ◆パソコンの周辺機器 ◆次のステップへ

PC-8001 RACET NECDOS 入門

塚越一雄著 1200円

●優秀な技術を誇るアメリカのソフトウェア・ハウスRACET COMPUTES Ltd. が開発したPC-8001向けDOS（ディスク・オペレーティング・システム）の入門書です。

【内容】 ◆RACETが日本に上陸する ◆OSとは何か ◆PC-8001のDOSをめぐる ◆RACET NECDOSとは ◆バックアップ作業 ◆PC-8001で立ち上げる ◆驚異のI/Oスルー ◆恐怖のMOUNT無用論 ◆ROM BASICはそのまま ◆プログラムをセーブする ◆ファイル・スペック ◆“color/bas”を走らせる ◆ファイルの削除・名称変更 ◆神秘的なファンクション・キー ◆“AUTOコマンドを使って ◆自動オペレーションの実現 ◆3本のプログラムが走る

PC-8001 はるみのゲームライブラリー

高橋はるみ著 1200円

●楽しいゲームソフトを25本収録しました。

【内容】 ◆いなばのしろうさぎ ◆水道管とねずみさん ◆DIVER ◆PANTHER
◆平安京スカナール ◆ボスコニアン ◆ラリー・X ◆とべとべこうもりくん
◆スペース・シップ ◆ギャング・ゲーム ◆ディック・マン ◆宇宙のゴミやさん
◆MARS ◆スキー・スラローム ◆パチンコ ◆はやうちマック ◆アップルアップル
◆トムとジェリー ◆QUICK ◆重戦車戦ゲーム ◆ペンキマン
◆てっぼううおくん ◆みつばちマーヤのぼうけん ◆オートレース

●別売「カセットテープ」 2500円 送料240円

PC-8001 はるみのゲームライブラリーPARTⅡ

高橋はるみ著 1200円

●1画面ソフトを63本収録してあります。

【内容】 ◆あるけあるけゲーム ◆スキースラローム ◆スペース・ストラクツ ◆ペン
ギンランド大混乱 ◆はやうちマック ◆キャノンF-2 ◆バイキンくんゲー
ム ◆テニス・ゲーム ◆スカッシュ ◆ブロックくずし ◆ジャンピング・エ
イリアン ◆UFOシューティング ◆チャイナシンドローム ◆エイリアンハ
ンター ◆クッション・ピンボール ◆みつばちマーヤの冒険 ◆ペンキマン
◆カーレース ◆地底探検ゲーム ◆レスキュー・ポート ◆ブラックホール
◆スクロールジャンプ ◆スペースアタッカー ◆サイモン ◆E.T. ◆アス
テロイドベルト ◆メテオ ◆サスケ

●別売「カセットテープ」 2500円 送料240円

PC-8001 はるみのプログラミングレッスン

高橋はるみ著 1200円

●ゲームを自分でつくってみたいという読者のために高橋さんがやさしく教えてくれます。ゲーム・プログラミングの基本がもりこまれています。

【内容】 ◆はじめましてマイコンくん！ ◆まちがったって気にしない ◆マイコンお
えかき教室 ◆マニュアルはすてきなおもちゃばこ ◆なんでもできるPRIN
Tくん◆いろんな画面を使っちゃおう！ ◆カラーいろいろ ◆線ひきじょう
ずのLINEくん ◆これがプログラム!? ◆はるみもプログラミング ◆だんだ
ん本格テキ！ ◆あるけあるけゲーム ◆SPACE STLUCTS ◆カーレース
◆ハッスル ◆もぐらたたき ◆マンホール ◆ファイヤー ◆SPACE ◆
タイプ練習

●別売「カセットテープ」 2500円 送料240円

PC-6001 はるみのゲームライブラリー

高橋はるみ著 1200円

●とびきりNOWなゲームソフトが20本もつまっています。

【内容】 ◆ルパン3世ゲーム ◆まんびき少年ゲーム ◆ギャラクシー・ウォーズ ◆ペンキマン ◆ギャング・ゲーム ◆スペース・シップ ◆スキースラローム ◆MARS ◆しばかりゲーム ◆宇宙のゴミやさん ◆トムとジェリー ◆はやうちマック ◆とべとべこうもりくん ◆AUTO RACE ◆みつばちマーヤのぼうけん ◆地雷突破作戦 ◆月面着陸 ◆アニマル・ハンター ◆平城京エイリアン ◆QUICK

●別売「カセットテープ」 定価2500円 送料240円

PC-6001 はるみのゲームライブラリーPARTⅡ

高橋はるみ著 1200円

●大好評のPARTⅠの続編です。マシン語なみのカラーグラフィックスをつかったパビコンゲーム決定版！

【内容】 ◆サスケ ◆ペンギン・ランド大混乱 ◆オリエンテーリング ◆ひねくれランナー ◆ブロックくずし ◆スーパーサイモン ◆タイプ練習 ◆地底探検ゲーム ◆ジャンピング・エイリアン ◆カーレース ◆キングコングJr. ◆UFOシューティング ◆オリエンテーリング1300 ◆E.T. ◆ビルディング・クラッシュ ◆チャイナ・シンドローム ◆クッション・ピンボール ◆おいかけマック ◆エイリアンハンター ◆平安京エイリアン

●別売「カセットテープ」 定価2500円 送料250円

PC-6001 はるみのプログラミング・レッスン

高橋はるみ著 1200円

●ちょっとパビコンになれてくると自分でもプログラムを作ってみたくなるもの。この本は、そんな読者のためのやさしいプログラミング教室です。

【内容】 ◆はじめましてマイコンくん ◆マニュアルはすてきなおもちゃばこ ◆カラいろいろ ◆線ひきじょうずなLINEくん ◆だんだん本格テキ！ ◆あるけあるけゲーム ◆まずは舞台づくりから ◆そしてエンディング！ ◆地雷突破作戦 ◆こんなゲームにしちゃうの！ ◆SPACE STLUCTS ◆ALIE N HUNTER ◆平安京エイリアン

●別売「カセットテープ」 定価2500円 送料240円

パソコン工作入門

安立隆一著 1200円

PC-6001, PC-8001, PC-8001MKII, PC-8801のユーザ向け手作りユニットの楽しみ方の本です。

【内容】 ◆PC-6001アダプタの製作 ◆PC-8001アダプタの製作 ◆PC-8801アダプタの製作 ◆ステレオシンセサイザの製作 ◆8チャンネルリレーユニットの製作 ◆16チャンネル電圧感知ユニットの製作 ◆多目的24ビットDIOユニットの製作 ◆音声合成おしゃべりユニットの製作 ◆SUPER EPROMライターの製作

インターフェース設計入門

●6月下旬発行

野崎 真著 2000円

【内容】 ◆インターフェースとは ◆論理回路の基礎 ◆インターフェース設計のための基礎知識 ◆入出力素子とそのインターフェース ◆マイクロプロセッサの動作 ◆I/Oインターフェースの手法 ◆PC8000/8800用インターフェースボードの設計 ◆RS232Cとそのインターフェースボード ◆IEEE-488とそのインターフェースボード ◆PC8000用インターフェースボード

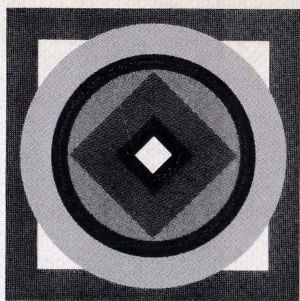
FM-7/FM-8 はるみのゲームライブラリー

高橋はるみ著 1200円

●楽しいゲームを20本収録しました。

【内容】 ◆スペース・ストラクツ ◆あるけあるけゲーム ◆スーパーサイモン ◆タイプ練習 ◆ビルディング・クラッシュ ◆MARS ◆しばかりゲーム ◆宇宙のゴミやさん ◆平安京エイリアン ◆水道管とねずみさん ◆いなばのしろうさぎ ◆キングコングJr. ◆E.T. ◆まんびき少年ゲーム ◆チャイナ・シンドローム ◆はやうちマック ◆みつばちマーヤの冒険 ◆オリエンテーリング1300 ◆ジャンピング・エイリアン ◆オートレース

●別売「カセットテープ」 定価2500円 送料240円



PC-9801

アセンブリ言語プログラミング入門[1]

[目次]より

ナツメ社
定価=1600円

0・0	アセンブリ言語とは?	8
0・2	フローチャート用紙とコーディング用紙	11
1・0	PC-9801のOS	18
1・3	メモリのセグメンテーション	22
1・5	機械語モニタの起動方法	26
2・1	8086の設計思想	33
2・3	セグメント・レジスタの役割	36
2・6	ポインタとインデックス・レジスタの役割	40
2・7	フラグの役割	41
3・0	メモリの内容をコピーするプログラム	44
3・2	メッセージを表示するプログラム	58
3・4	16ビットの加減算を行なうプログラム	65
3・6	論理演算をテストするプログラム	76
3・7	16ビットデータを10倍するプログラム	82
3・8	サブルーチンの使い方	85
3・9	スタックの概念とレジスタ待避	90
4・1	ゼロ・フラグ<ZF>による分岐	113
4・2	キャリー・フラグ<CF>による分岐	118
5・0	くり返しのあるプログラム	128
5・6	簡単なデモンストレーション	147
5・7	ローテートおよびシフトの概念	150
6・0	8086のインストラクション・セット	159
6・1	データ転送命令	162
6・3	乗除算命令	168
6・5	ストリング操作命令	172
6・6	分岐命令	175
7・0	8086のインストラクション活用表	182

